ENSIAME

Optimisation:

De l'estimation paramétrique à l'apprentissage, une ballade entre théorie et pratique

S. Delprat

Chapitre 6 – Réseaux de neurones

Et le modèle dans tout ça ?

Les outils d'optimisation permettent de minimiser une fonction. Les applications sont nombreuses pour peu qu'on dispose d'un modèle du procédé à manipuler.

Il existe des cas où les équations de la physique sont trop complexes pour être exploitables.

On utilise alors une approche « boite noire »: on impose :

- une structure de modèle : polynôme d'ordre n, spline, etc.
- Un nombre de paramètres libres (coef du polynôme, etc)

Reste alors le problème de l'estimation paramétrique: trouver la valeur des paramètres qui minimise une erreur de modèle.

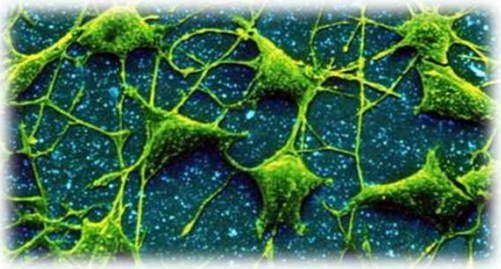
=> Les réseaux de neurones permettent de représenter une grande variété de modèles

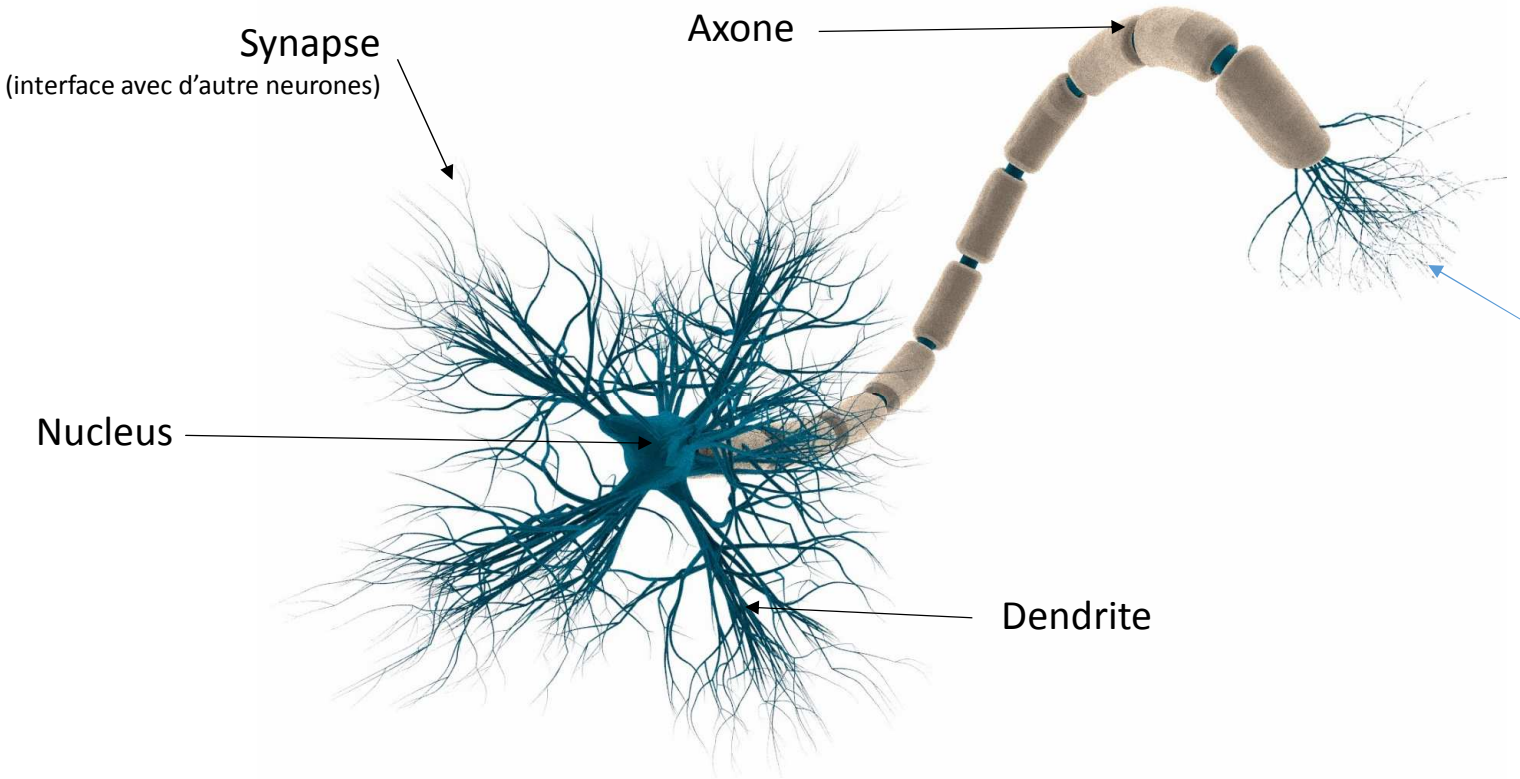
1) Réseau de neurones
2) Apprentissage des poids
3) Différents types de réseaux

Introduction

<u>Rappel de biologie:</u>

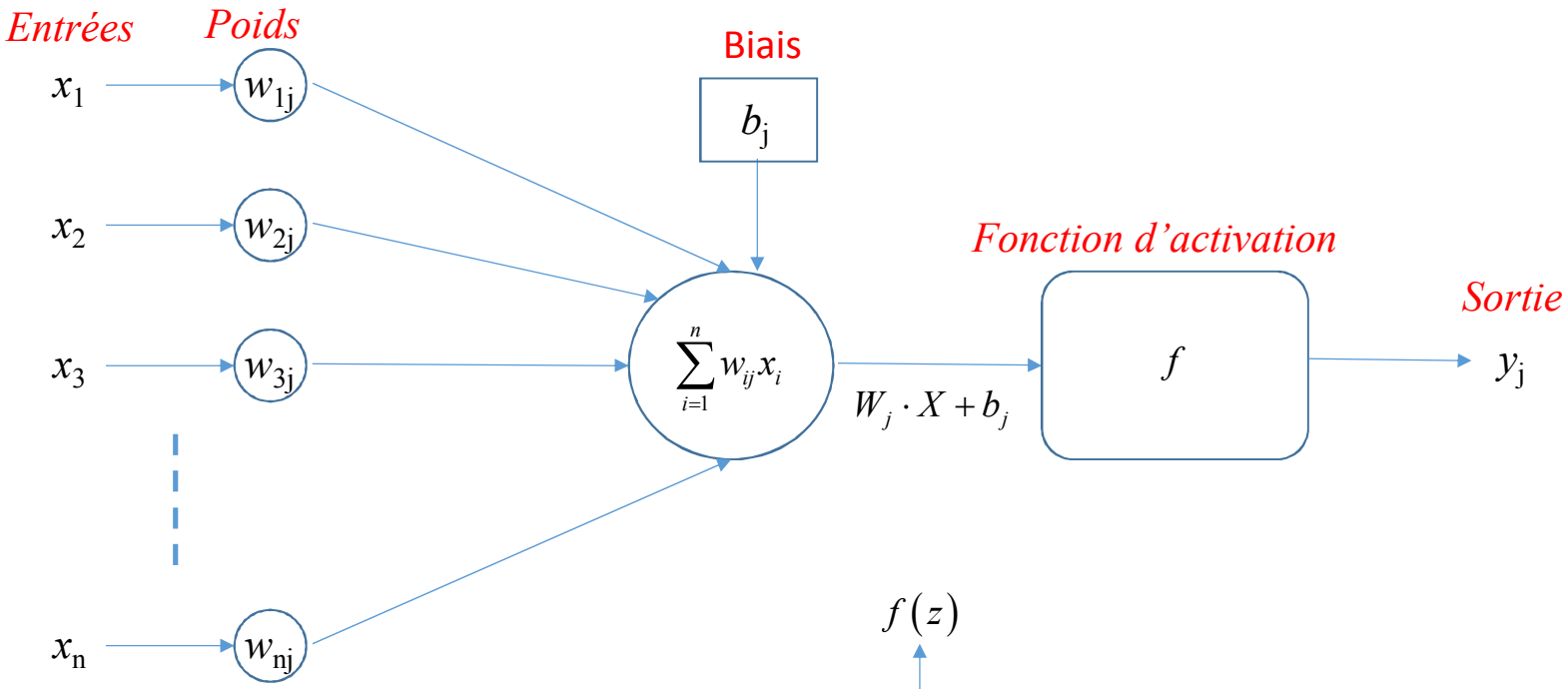
Les cerveaux (humains, animaux) sont constitués d'un assemblage de neurones.





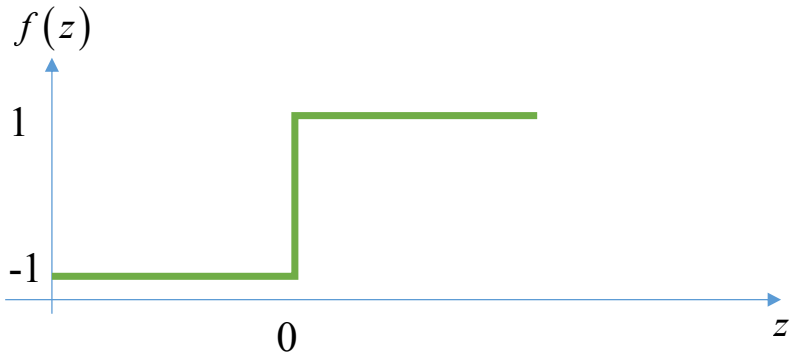
Perceptron

Modèle informatique d'un neurone : Le perceptron



Fonction d'activation du perceptron:

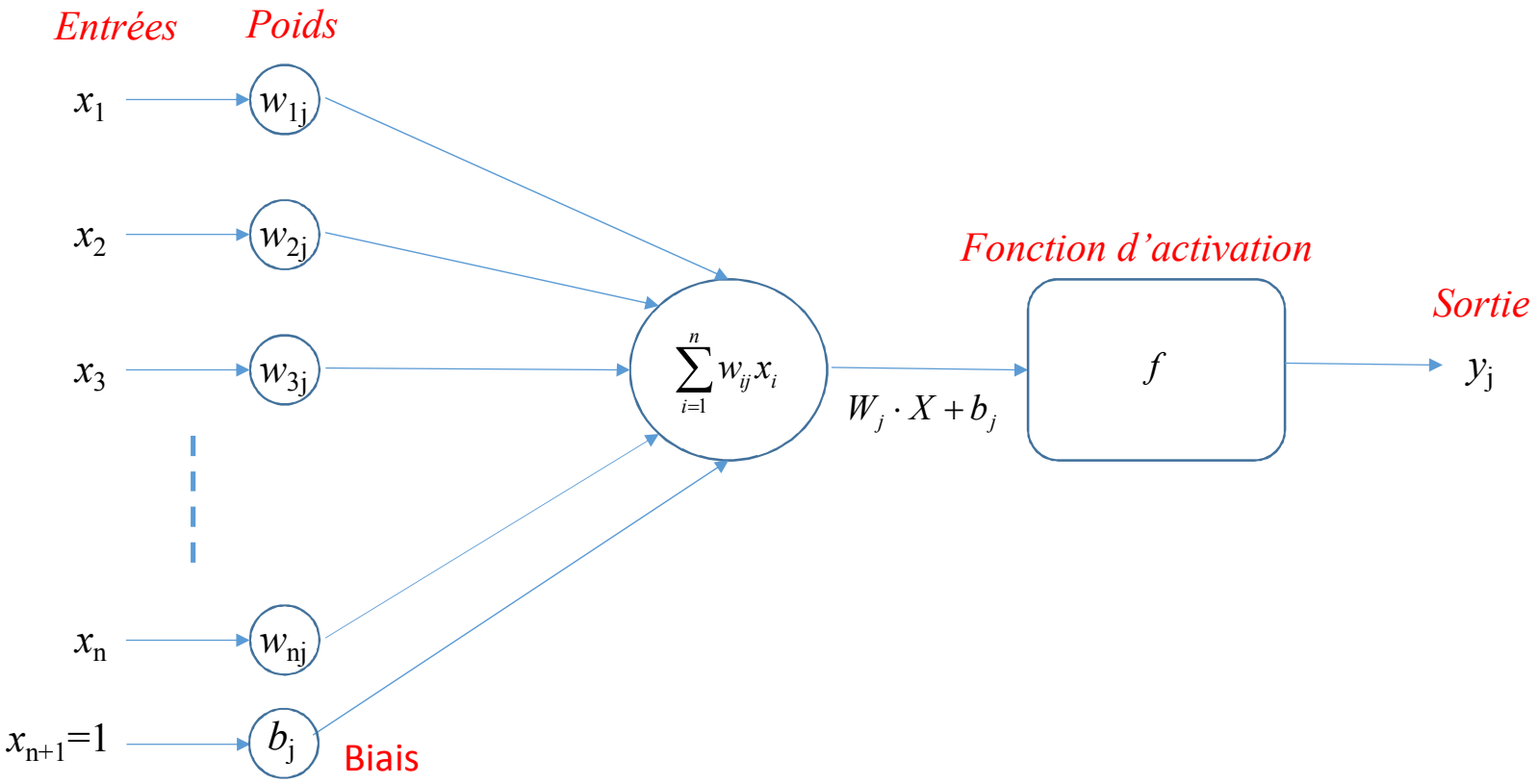
$$f(z) = \begin{cases} -1 & z < 0 \\ 1 & z \geq 0 \end{cases}$$



Perceptron

Modèle informatique d'un neurone : Le perceptron

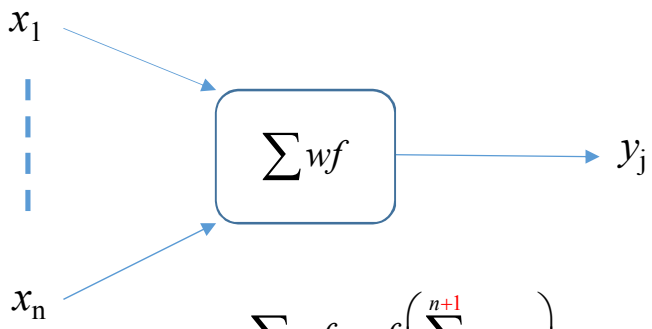
En pratique, on peut simplifier les programmes en imposant : $x_{n+1}=1$ et $w_{n+1,j}=b_j$



Perceptron

Modèle informatique d'un neurone : Le perceptron

Le neurone artificiel peut également être représenté de manière compacte

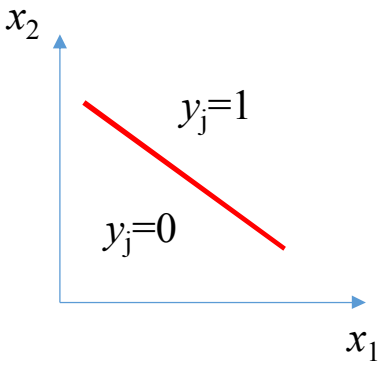


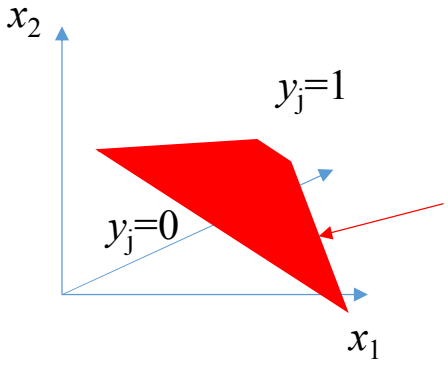
$$\sum wf = f\left(\sum_{i=1}^{n+1} w_{ij} x_i\right)$$

Intègre le biais dans les poids w_i

Perceptron

Le perceptron effectue une classification linéaire

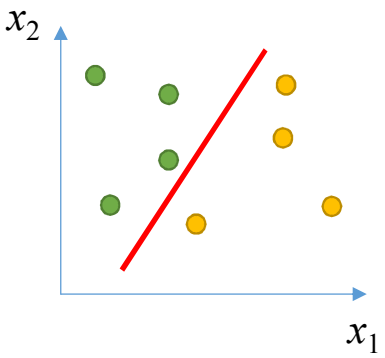




Hyperplan de séparation

L'hyperplan de séparation ne dépend que des poids w_i du neurone.

Le problème de classification est donc de trouver les poids w_i qui séparent correctement les données en 2 classes



x_1	x_2	Classe
1	3	0
3	2	1
2,5	1	1
1	0,5	0

Perceptron

Soit $X_i \in \mathbb{R}^n$ une entrée $X_i = \left\{x_i^1, x_i^2, \cdots, x_i^n\right\}$

On définit 2 ensembles, associés aux classes 0 et 1 : $P_0 = \left\{X_i \in \mathbb{R}^n, i = 1..m \mid classe(X_i) = 0\right\}$

$P_1 = \left\{X_i \in \mathbb{R}^n, i = 1..m \mid classe(X_i) = 1\right\}$

On suppose que les entrées sont linéairement séparable (i.e. que le problème admet une solution).

L'algorithme d'apprentissage des poids est le suivant:

W=0; b=1;

Tant que la classification n'est pas correcte

Pour i=1 à m

Si $f\left(WX_i + b\right) \neq C_i$

Si Ci>0 ALORS

$W = W + X_i; b = b + 1$

SINON

$W = W - X_i; b = b - 1$

Fin SI

Adaptation des poids en fonction de l'erreur commise

Fin Pour

Fin While

L'algorithme converge si les données sont exactement linéairements séparables

Perceptron

```
clear all;
close all;
clc;
rng(3,'twister');
% Générate a dataset
n=2;
m=100;
[X,C]=GenereExemple(n,m);

% Apprentissage
iP0= C==0;
iP1= C==1;
P0=X(:,iP0);
P1=X(:,iP1);

% Initialisation des poids
W=zeros(n,1);
b=0;
```

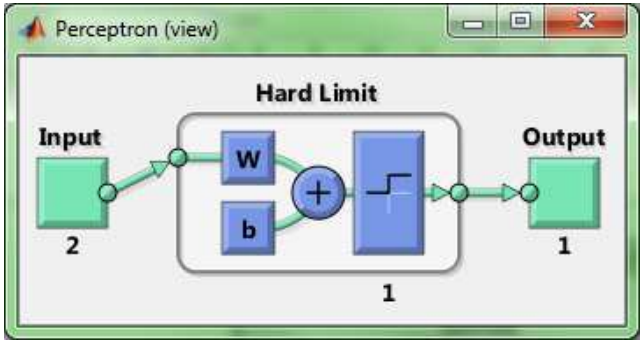
```
% Aprrentissage
nErreur=EvaluePerformance(X,C,W,b,m);
k=1;
while nErreur>0
    fprintf('Iteration : %i Erreur : %i \n',k,nErreur);
    for i=1:m
        if (W'*X(:,i)+b>0)~=C(i)
            if C(i)==1
                W=W+X(:,i);
                b=b+1;
            else
                W=W-X(:,i);
                b=b-1;
            end
        end
    end
    nErreur=EvaluePerformance(X,C,W,b,m);
    k=k+1;

    a=-(W(1)+b)/W(2);
    figure(10);
    clf;
    plot(P0(1,:),P0(2,:),'r*');
    hold on;
    plot(P1(1,:),P1(2,:),'bo');
    grid on;
    hold on;
    xx=xlim;
    plot(xx,-(W(1)*xx+b)/W(2),'r','linewidth',2);
    drawnow;
end
```

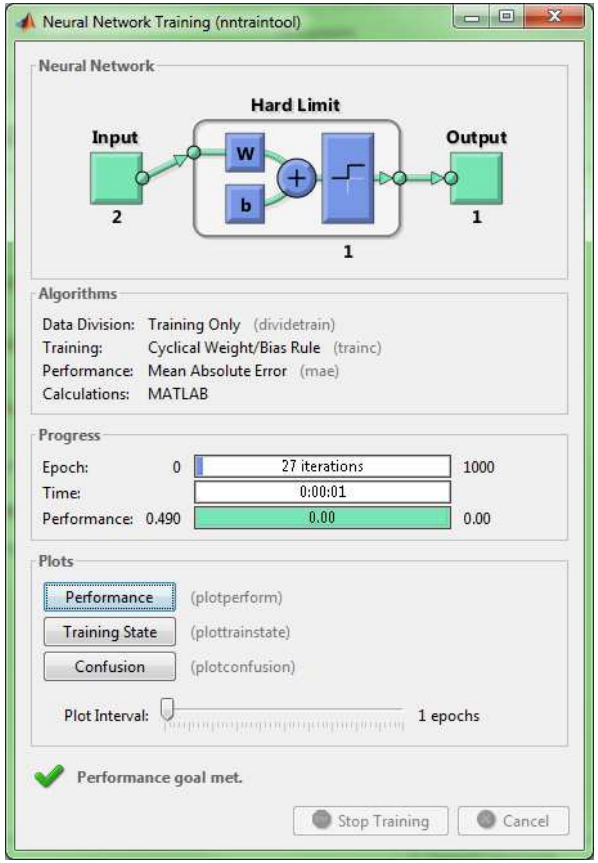
```
function Erreurs=EvaluePerformance(X,C,W,b,m)
Erreurs=0;
y=(W'*X+b)>0; % Sortie du perceptron
Erreurs=sum(C~=y);
end
function [X,C]=GenereExemple(n,m)
X=rand(n,m);
A=(rand(1,n)-.5)*10;
B=-mean(A*X);
C=A*X+B>0;
end
```

Perceptron

Le même résultat peut être obtenu avec la Neural Network toolbox

```
clear all;
close all;
clc;
rng(3,'twister');
% Générate a dataset
n=2;
m=100;
[X,C]=GenereExemple(n,m);

net = perceptron;
net = train(net,X,C);
view(net)
y = net(X);
fprintf('Nombre d''erreurs : %i\n',sum(y-C))

function [X,C]=GenereExemple(n,m)
X=rand(n,m);
A=(rand(1,n)-.5)*10;
B=-mean(A*X);
C=A*X+B>0;
end
```

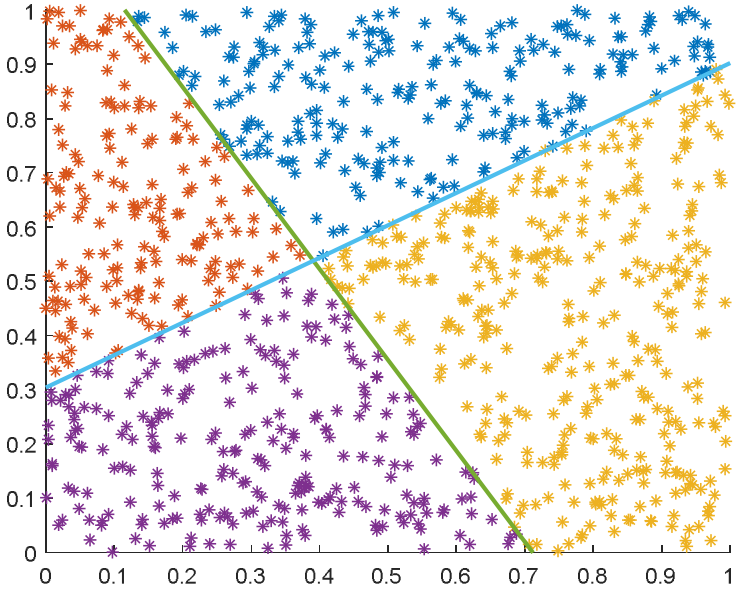




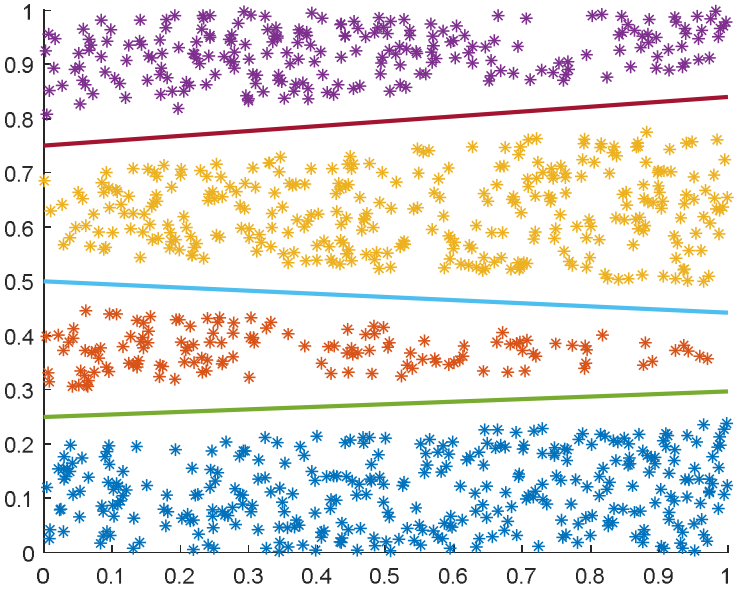
Perceptron *multi-classes*

Problème multi-classes:

Chaque classe n'est pas linéairement séparable de toutes les autres



Chaque classe est linéairement séparable de toutes les autres => ok

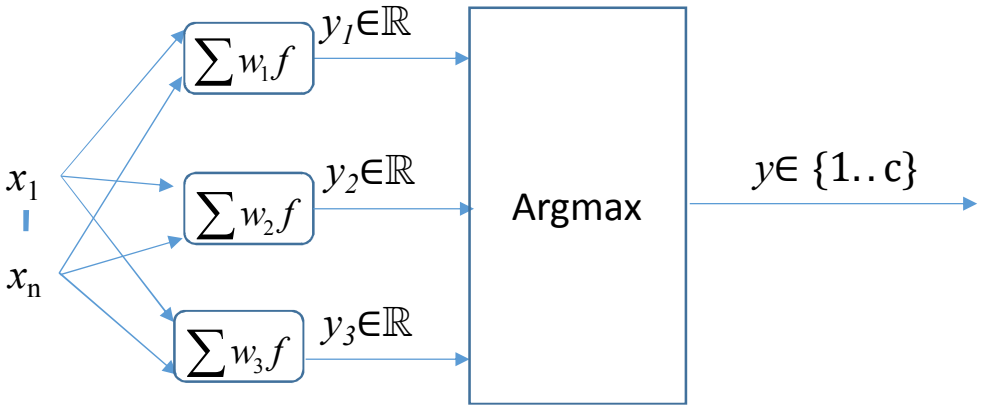


Perceptron *multi-classes*

Comment faire si on dispose de $c>2$ classes ?

Approche n°1 : One against ALL (OA)

Fonction d'activation linéaire: $f(z)=z$



Les poids doivent être adaptés de telle sorte que la sortie de chaque réseau apprenne à identifier une classe

Perceptron *multi-classes*

```
clear all;
close all;
clc;

rng(2,'twister');

n=2;   % Nombre de features
m=1000; % Nombre d'individus
c=7; % Nombre de classe
[X,C,B,A,We]=GenereExemple(n,m,c);

figure(10);
hold on;
for i=1:c
    plot(X(1,C==i),X(2,C==i),'*');
end
xx=xlim;
for i=1:length(A)
    plot(xx,A(i)*xx+B(i),'linewidth',2);
end

% Apprentissage
W=zeros(c,n+1);
[nErreur,yest]=EvaluePerformance(X,C,W,m);
k=1;
while nErreur>0
    fprintf('Iteration : %i Erreur : %i \n',k,nErreur);
    for i=1:m
        x=X(:,i)';
        % Estimation de la sortie
        if yest(i)~=C(i)
            % Prédiction incorecte
            iOk=1:c==C(i);
            inotok=1:c~=C(i);
            W(iOk,:)=W(iOk,:)+x;
            W(inotok,:)=W(inotok,:)-x;
        end
    end
    [nErreur,yest]=EvaluePerformance(X,C,W,m);
    figure(100);
    hold on;
    plot(k+W(:)*0,W(:),'*');
    drawnow;
    k=k+1;
end
title(sprintf('Iteration : %i Erreur : %i \n',k,nErreur));
```

Perceptron *multi-classes*

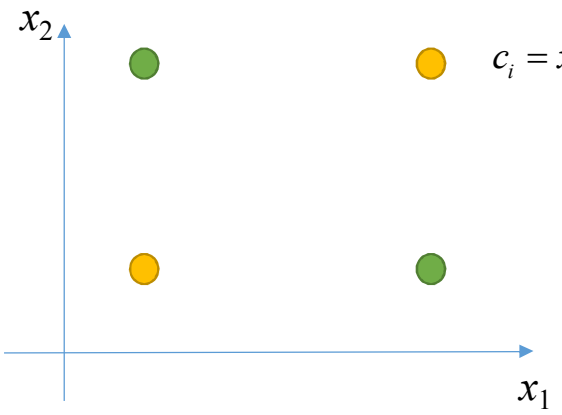
```
function [X,C,B,A,W]=GenereExemple(n,m,nclasses)
% Génère des valeurs de classes entre 1 et nclasses
X=rand(n,m);
X=[X;ones(1,m)]; % Rajoute 1 pour le biais
C=NaN(1,m);
for i=1:nclasses-1
    Delta=1/(nclasses); % Largeur de bande
    A(i)=(rand-.5)*Delta;
    B(i)=Delta*i;
    W(i,:)=[A(i) -1 B(i)];
end
ended=0;
while ended==0
    b=abs(W*X./W(:,2));
    inotok=logical(sum(b<Delta/5));
    X(1:2,inotok)=rand(2,sum(inotok));
    ended=all(~inotok);
end
i1=W(1,:)*X>0;
C(i1)=1;
for i=2:nclasses-1
    iok=(W(i-1,:)*X<0)&(W(i,:)*X>0);
    C(iok)=i;
end
i=nclasses-1;
i1=W(i,:)*X<0;
C(i1)=nclasses;
end
```

```
function [Erreurs,y]=EvaluePerformance(X,C,W,m)
Erreurs=0;
[~,y]=max(W*X);
Erreurs=sum(C~=y);
end
```

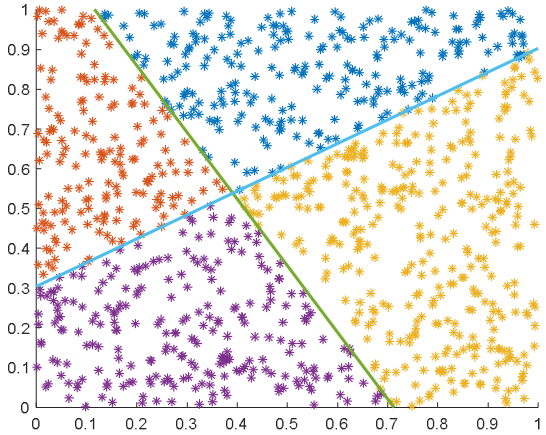
Perceptron

Le perceptron est très limité.

Par exemple, il est impossible de classifier la fonction XOR ou ces données

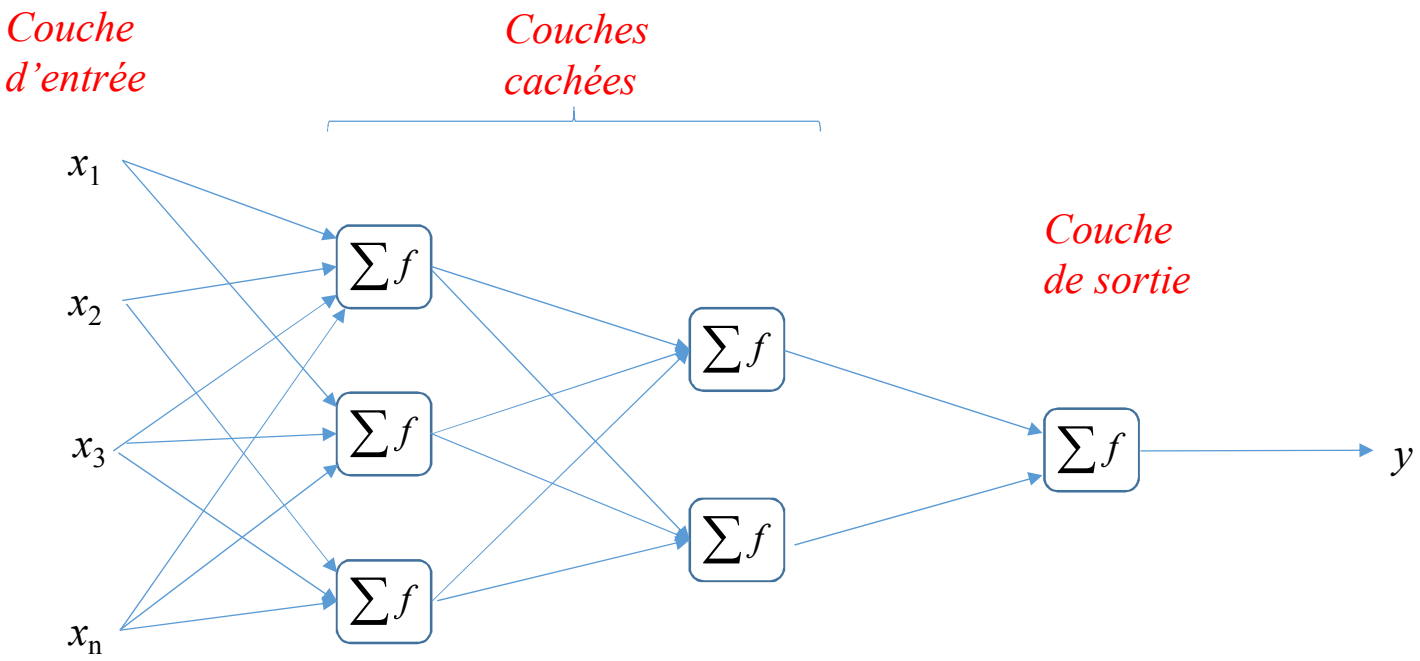


$$c_i = xor\left(x_i^1, x_i^2\right) \in \{0,1\}$$



Une solution est de rajouter des couches intermédiaires.

Réseau de neurones - *structure*



Le nombre et la taille des couches cachées peut être quelconque.

Réseau de neurones – *Fonctions d'activation*

En général, il est préférable d'utiliser des fonctions d'activations dérivables.

	Fonction	Dérivée	Nom Matlab
• Identité :	$f(z)=z$	$f'(z)=1$	purelin
• Logistique / Sigmoïde :	$f(z)=\frac{1}{1+e^{-z}}$	$f'(z)=f(z)\left(1-f(z)\right)$	logsig
• Tangente hyperbolique :	$f(z)=\tanh(z)$	$f'(z)=1-f(z)^2$	tansig
• Arc-tangente:	$f(z)=\tan^{-1}(z)$	$f'(z)=\frac{1}{z^2+1}$	
• Rectification linéaire (ReLu)	$f(z)=\begin{cases}0 & z<0\\ z & \text{sinon}\end{cases}$	$f'(z)=\begin{cases}0 & z<0\\ 1 & \text{sinon}\end{cases}$	poslin

Cf https://fr.wikipedia.org/wiki/Fonction_d%27activation pour une liste plus complète

Réseau de neurones – *Fonctions d'activation*

En général, il est préférable d'utiliser des fonctions d'activations dérivables.

	Fonction	Dérivée	Nom Matlab
• Identité saturée:	$f(z)=\begin{cases}0 & z\le 0\\ z & 0<z\le 1\\ 1 & z>1\end{cases}$	$f'(z)=\begin{cases}0 & z\le 0\\ 1 & 0<z\le 1\\ 0 & z>1\end{cases}$	satlin
• Echelon	$f(z)=\begin{cases}0 & z\le 0\\ 1 & z>1\end{cases}$	$f'(z)=\delta(z)$	hardlim

Attention : certaines fonctions ne sont disponibles qu'avec certains type de couches.

Réseau de neurones – *Fonctions d'activation*

La structure présentée permet d'approcher une vaste classe de fonctions $y = f(x) \quad f : \mathbb{R}^m \to \mathbb{R}^p$
et donc de traiter différents types de problèmes.

Si les sorties $y \in \mathbb{N}^p$ sont à valeurs discrètes alors il s'agit d'un problème de classification.

Si les sorties $y \in \mathbb{N}^p$ sont réelles alors il s'agit d'un problème d'approximation.

Réseau de neurones – *Fonctions d'activation*

```
clear all;
close all;
clc;
% Y=A.X+B <=> W1*X1+W2*X2+W3=0
% avec W2=-1; W1=A; W3=B

% 2 Perceptron à 1 couche, pour 4 classes
rng(20,'twister');

% Générate a linear separable dataset
n=2;   % Nombre de features
m=500; % Nombre d'individus
c=8;   % Nombre de classes

% [X,C,B,A,We]=GenereExemple(n,m,c);
[X,C,B,A,We]=GenereExemple2(n,m,c);

Inputs=X(1:2,:);
Outputs= C;

hiddenSizes=2;
net=feedforwardnet(hiddenSizes);
```

Crée un réseau multi-couches
1 couche taille 2

Réseau de neurones – *Fonctions d'activation*

```
net.Layers{1}.transferFcn='tansig';
```

Paramétrage des couches
(d'autres paramètres disponibles)

```
net.divideParam.trainRatio = 1; % training set [%]
net.divideParam.valRatio   = 0; % validation set [%]
net.divideParam.testRatio  = 0; % test set [%]
net.adaptFcn='learnpn';
```

Paramétrage de l'algorithme d'apprentissage

```
net=configure(net,Inputs,Outputs);
[net,tr,Y,E] = train(net,Inputs,Outputs);
%view(net);
```

Apprentissage du réseau sur les données fournies

```
yest=round(net(Inputs));
fprintf('Nombre d''erreurs : %i\n',sum(yest~=C));
```

Evaluation du réseau sur les entrées Inputs

Réseau de neurones – *Fonctions d'activation*

```
[X,Y]=meshgrid(linspace(0,1,100),linspace(0,1,101));
inp=[X(:)';Y(:)'];
Z=zeros(size(X));
Z(:)=net(inp);
```

Génération d'une grille X-Y
Et évaluation du réseau de neuronnes

```
figure(10);
subplot(1,2,1);
hold on;
for i=1:c,plot(Inputs(1,C==i),Inputs(2,C==i),'*');end
xx=xlim;yy=ylim;
for i=1:length(A)
    plot(xx,A(i)*xx+B(i),'linewidth',2);
end
xlim([0 1]);ylim([0 1]);
subplot(1,2,2);surf(X,Y,Z,'EdgeColor','none');grid on;
hold on;
hold on;
for i=1:c
    plot3(Inputs(1,C==i),Inputs(2,C==i),C(C==i),'*');
end

function [Erreurs,y]=EvaluePerformance(X,C,W,m)
Erreurs=0;
[~,y]=max(W*X);
Erreurs=sum(C~=y);
end
```

Réseau de neurones – *Fonctions d'activation*

Fonctions permettant de générer aléatoirement des données

```
function [X,C,B,A,W]=GenereExemple(n,m,nclasses)
% Génère des valeurs de classes entre 1 et nclasses
X=rand(n,m);
X=[X;ones(1,m)]; % Rajoute 1 pour le biais
C=NaN(1,m);
for i=1:nclasses-1
    Delta=1/(nclasses); % Largeur de bande
    A(i)=(rand-.5)*Delta;
    B(i)=Delta*i;
    W(i,:)=[A(i) -1 B(i)];
end
i1=W(1,:)*X>0;
C(i1)=1;
for i=2:nclasses-1
    iok=(W(i-1,:)*X<0)&(W(i,:)*X>0);
    C(iok)=i;
end
i=nclasses-1;
i1=W(i,:)*X<0;
C(i1)=nclasses;
end
```

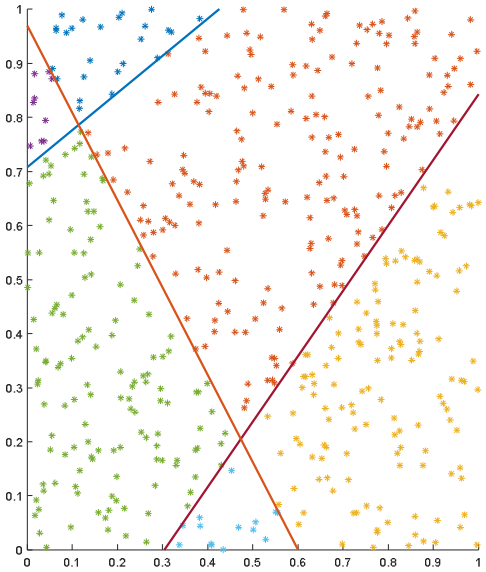
```
function [X,C,B,A,W]=GenereExemple2(n,m,nclasses)
% Génère des valeurs de classes entre 1 et nclasses
iter=nextpow2(nclasses);
if nclasses~=2^iter
    error('nclasses doit être de la forme 2^m');
end
ended=0;
while ended==0
    X=rand(n,m);
    X=[X;ones(1,m)]; % Rajoute 1 pour le biais

    C=ones(1,m);
    for i=1:iter
        % Droite qui passe par (x1,y1) et (x2,y2)
        x1=mod(i,2)*rand;
        y1=rand*0.5+0.25;
        x2=rand*0.5+0.25;
        y2=mod(i+1,2)*rand;
        A(i)=(y1-y2)/(x1-x2);
        B(i)=y1-A(i)*x1;
        W(i,:)=[ A(i) -1 B(i)]/rand;
        C=C+(W(i,:)*X>0)*2^(i-1);
    end
    for i=1:nclasses
        cumC(i)=sum(C==i);
    end
    ended=1;
end
end
```

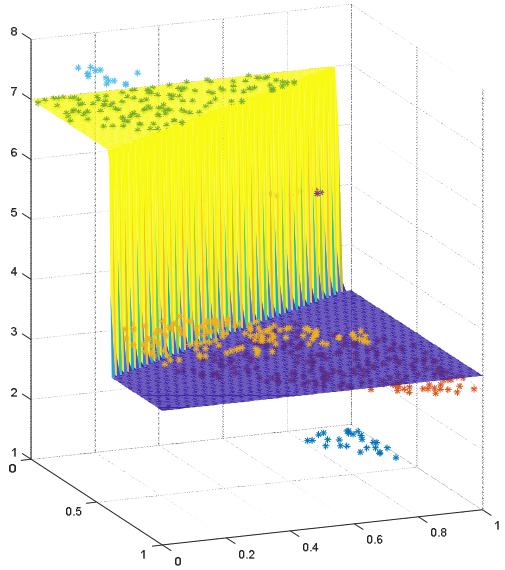
Réseau de neurones – *Fonctions d'activation*

Exemple : 8 classes, 500 données 1 couche de taille 1

=> 182 erreurs

=> Ne fonctionne pas car le réseau ne peut pas générer suffisamment de discontinuités



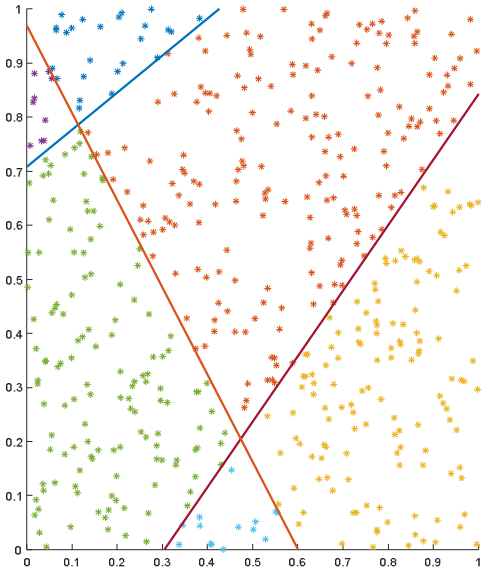


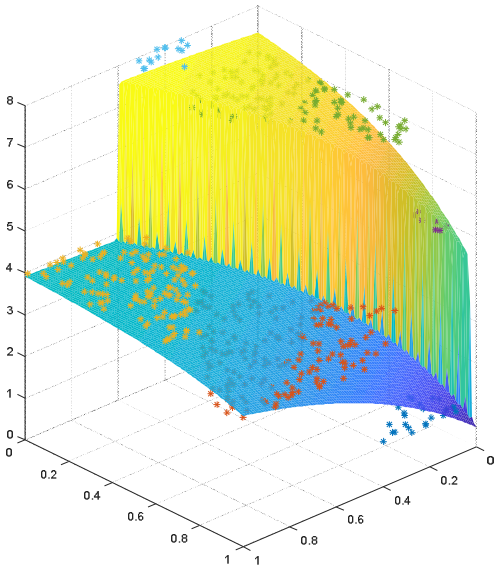
Réseau de neurones – *Fonctions d'activation*

Exemple : 8 classes, 500 données 1 couche de taille 2

=> 85 erreurs

=> Ne fonctionne pas car le réseau ne peut pas générer suffisamment de discontinuités

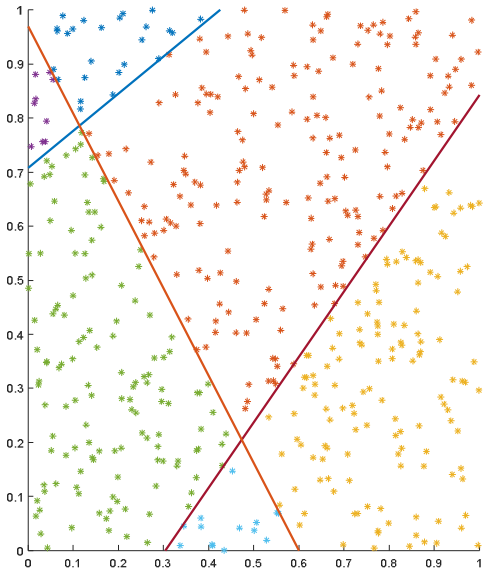


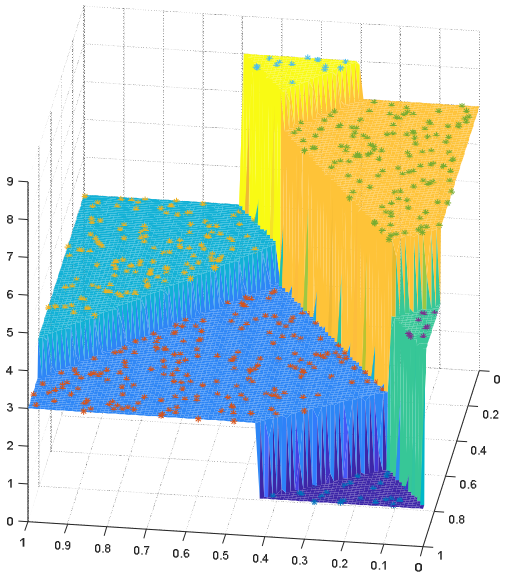


Réseau de neurones – *Fonctions d'activation*

Exemple : 8 classes, 500 données 1 couche de taille 3

=> 0 erreurs
=> Fonctionne car le réseau ne peut générer 2^3 = 8 classes





1) Réseau de neurones
2) Apprentissage des poids
3) Différents types de réseaux

Réseau de neurones – *Apprentissage*

On distingue plusieurs approches pour le calcul des poids d'un réseau de neurones.

Apprentissage supervisé:

On dispose de données représentatives ainsi qu'une réponse attendue pour ce problème.
L'objectif est de trouver les poids du réseau de telle sorte à minimiser l'erreur entre les données connues et la sortie du modèle.
Exemple d'application:

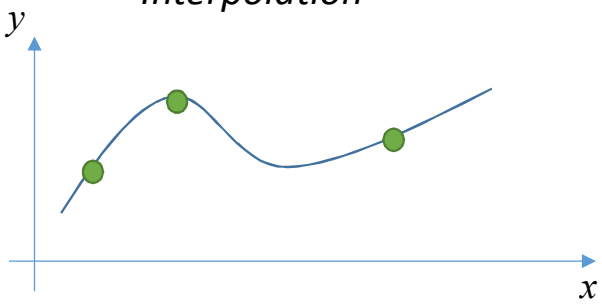
- Régression : c'est une compression de données. Au lieu de manipuler m>>0 données, on garde en mémoire uniquement les poids du réseau. On stocke donc moins de données
- Interpolation: On cherche à savoir ce qu'il se trouve entre 2 points mesurés. Nécessite des hypothèses sur le modèle
- Classification: On cherche à grouper les données dans des classes

Données: $\{x_i, y_i$,i= 1..$10^6\}$

Poids: $\{w_i$, i= 1..$10^2\}$

$$y = f(x, w)$$

Interpolation



Classification

Homme

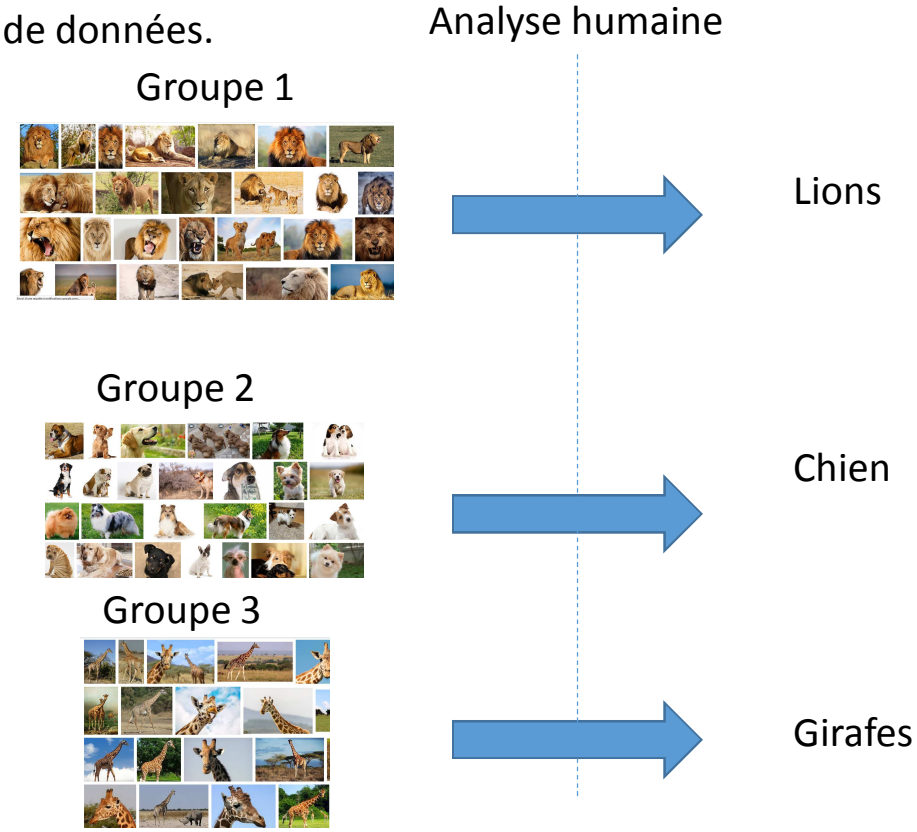
Femme

Réseau de neurones – *Apprentissage*

On distingue plusieurs approches pour le calcul des poids d'un réseau de neurones.

Apprentissage non supervisé:

L'objectif principal est d'expliquer un ensemble de données.
Schématiquement, il s'agit de faire des groupes de données.



Réseau de neurones – *Apprentissage*

On distingue plusieurs approches pour le calcul des poids d'un réseau de neurones.

Apprentissage semi supervisé ou réenforcement :

On ne dispose d'aucun jeu de mesure a priori.
Le logiciel d'apprentissage propose une solution. Un système évalue (note) la solution proposée.
C'est une méthode d'essais erreur qui permet d'adapter les poids en fonction du résultat d'une expérience.

Un exemple d'algorithme : Q-learning

Réseau de neurones – *Apprentissage supervisé*

Matlab propose l'ensemble des fonctions nécessaires à l'apprentissage.

Il s'agit en fait de résoudre un problème d'optimisation: $\min_W C = \sum_{i=1}^{n_x} \|E_i\| \qquad E_i = \hat{y}(X_i, W) - y_i$

$X_i \in \mathbb{R}^n \quad i = 1 \cdots n_x$: entrées

$\hat{y}(X_i, W) \in \mathbb{R}^p$: sorties estimées

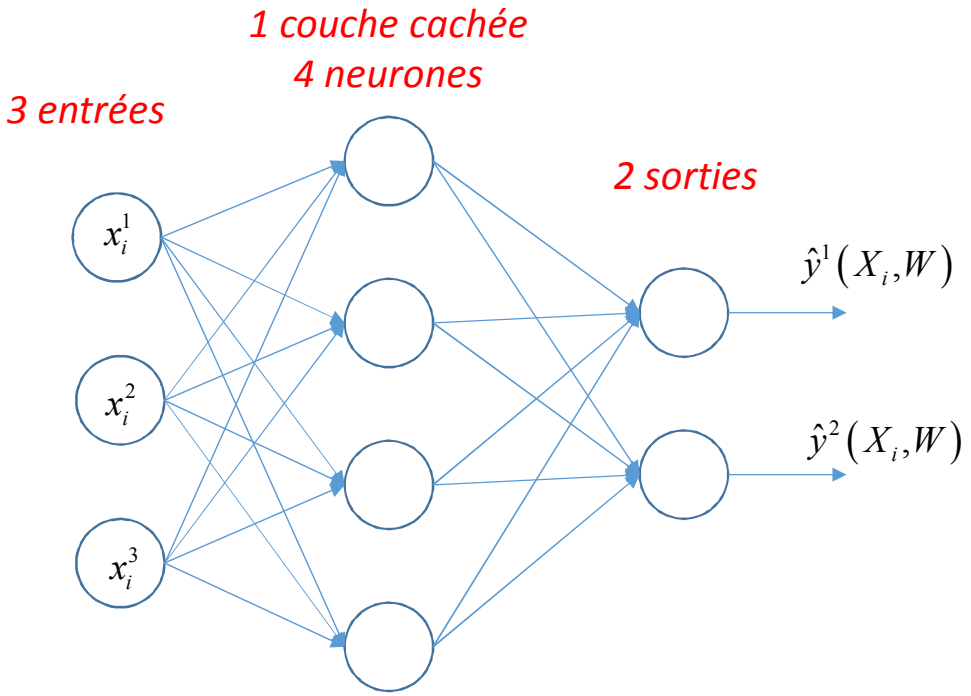
$W \in \mathbb{R}^{n_w}$: poids (paramètres du réseau de neurones)

$\| \ \|$: norme permettant de mesurer le résidu

Quand il y a plusieurs sorties, on décompose le critère:

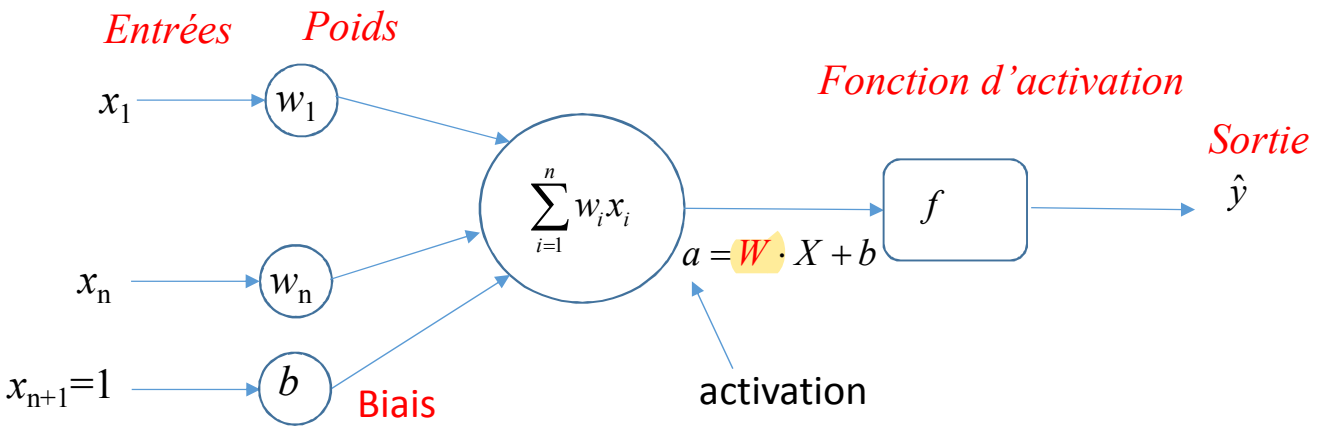
$$\min_W C = \sum_{j=1}^{p} C_j \quad E_j = \frac{1}{2n_x} \sum_{j=1}^{n_x} \left(\hat{y}^j(X_i, W) - y_i^j \right)^2$$

=>Problème pouvant être résolu en utilisant une grande variété d'algorithmes : SQP, Gradient Conjugués, Trust region, Algorithmes génétiques



Un réseau de neurones n'est en réalité qu'une « grosse » fonction non linéaire => **<u>on peut calculer son gradient</u>**

Réseau de neurones – *Apprentissage supervisé*



La variable optimisée est le vecteur des poids $W=(w_i)$. Ce n'est pas x

Le critère à optimiser n'est pas la fonction d'activation f mais le critère C.

Pour 1 seul neuronne:

Les entrées : $X_i = \left(x_i^j\right) \in \mathbb{R}^m \quad i = 1..n_x \quad j = 1..m$ n_x : nombre d'exemples m : taille de chaque exemple

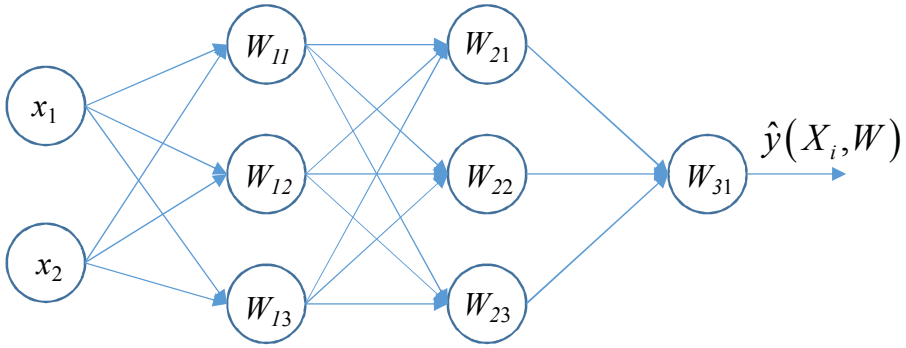
La sortie $y_i = f\left(W \cdot X_i + b\right) \quad W = \left(w_k\right) \in \mathbb{R}^{n_w} \quad b \in \mathbb{R}$ n_w : nombre de poids

L'erreur $E = \frac{1}{2n_x} \sum_{i=1}^{n_x} \left(f\left(W \cdot X_i + b\right) - y_i\right)^2$ $y_i \in \mathbb{R}$ Valeur cible pour le ième exemple

Gradient de l'erreur : $\frac{\partial E}{\partial w_k} = \frac{1}{n_x} \sum_{i=1}^{n_x} \left[\left(f\left(W \cdot X_i + b\right) - y_i\right) \cdot \frac{\partial f\left(W \cdot X_i + b\right)}{\partial w_k} \right]$

Réseau de neurones – *Apprentissage supervisé*

Cas d'un réseau de neurones:



Chaque neurone dispose de poids : $W_{ij} = \left(w_{ij}^k\right)$

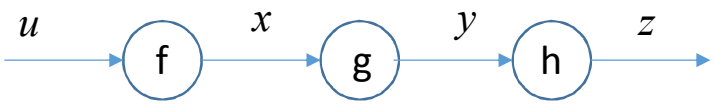
Erreur : $E = \frac{1}{2n_x}\sum_{i=0}^{1}\left(\hat{y}(X_i, W) - y_i\right)^2$

Le gradient :

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{1}{n_x}\sum_{i=1}^{n_x}\left(\hat{y}(X_i, W) - y_i\right)\frac{\partial \hat{y}(X_i, W)}{\partial w_{ij}^k}$$

Le problème est donc de comprendre comment une variation d'un des poids se propage dans le réseau pour finalement modifier l'erreur.

Analytiquement, cela se calcule en utilisant la dérivation en chaine car le réseau est essentiellement une fonction composée:



$$z = h \circ g \circ d(u)$$

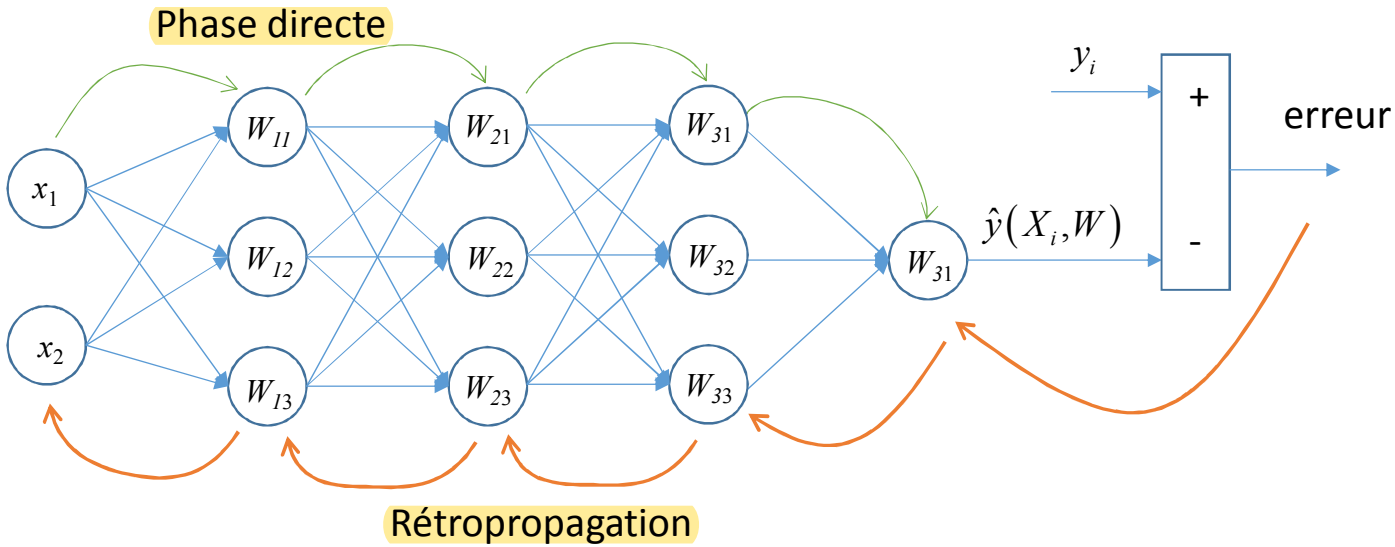
$$\frac{\partial z}{\partial u} = \frac{\partial h(g)}{\partial g} \cdot \frac{\partial g(d)}{\partial d} \cdot \frac{\partial d}{\partial u}$$

L'algorithme n'est pas décrit dans ce cours. Il n'est pas très difficile mais requiert des notations complexes.
Il est appelé: « Rétropropagation du gradient de l'erreur » ou « back propagation ».

Réseau de neurones – *Dilution du gradient*

De manière très schématique, l'apprentissage se fait en 2 passe:

- Passe directe (forward) : A partir des images, on calcule la sortie
- Passe arrière (backward) : l'erreur obtenu est « renvoyée » de la fin du réseau vers la première couche

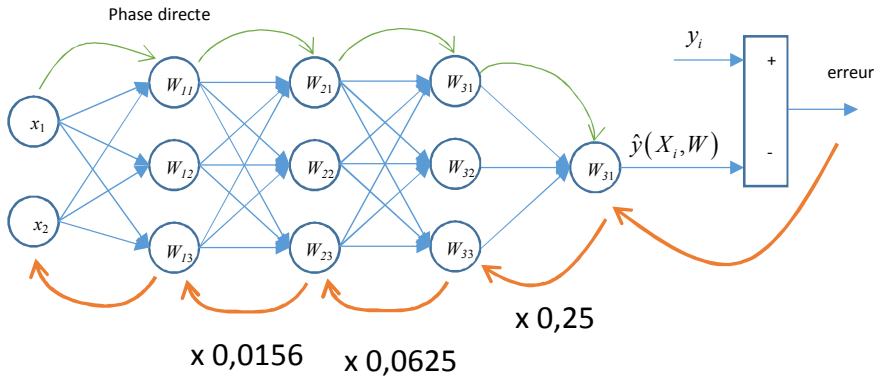


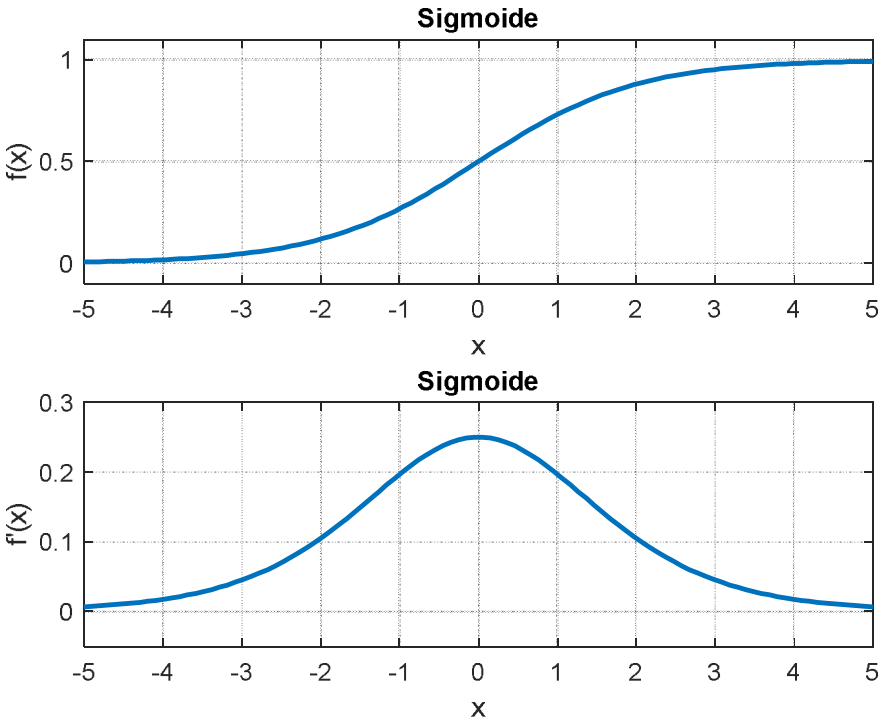
La phase de rétropropagation permet d'adapter les poids en fonction de l'erreur. La méthode se base sur le gradient de l'erreur.

Réseau de neurones – *Dilution du gradient (Vanishing gradient)*

Pendant la phase de rétropropagation, l'erreur propagée passe dans la dérivée de la fonction d'activation.
La sigmoide a un maximum inférieur à 0,25 => à chaque couche l'amplitude de l'erreur propagée est divisée au moins par 4

Nombre de couches traversée	1	2	3	4
Borne max sur l'amplitude pour une erreur de 1	0,25	0.0625	0.0156	0.0039



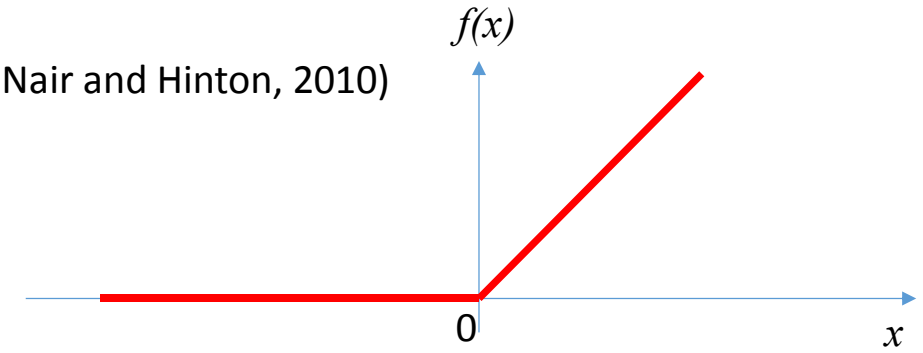


⇒ l'erreur propagée ne permet pas d'adapter rapidement les poids des premières couches
⇒ Limite l'utilisation à 2 – 3 couches maximum

Réseau de neurones – *Dilution du gradient (Vanishing gradient)*

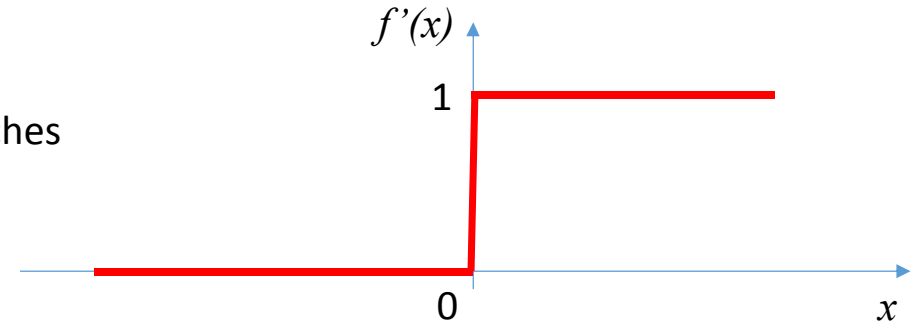
Comment luter contre la dilution du gradient ?
=> Utiliser une fonction d'activation non bornée : REctified Linear Unit (Nair and Hinton, 2010)

ReLU $$f(x) = \begin{cases} 0 & si \quad x < 0 \\ 1 & \text{sinon} \end{cases}$$



=> Donne de bons résultats en pratique.
=> Explication bio-inspirée

En pratique, très fortement conseillé pour des réseaux de plus de 3 couches



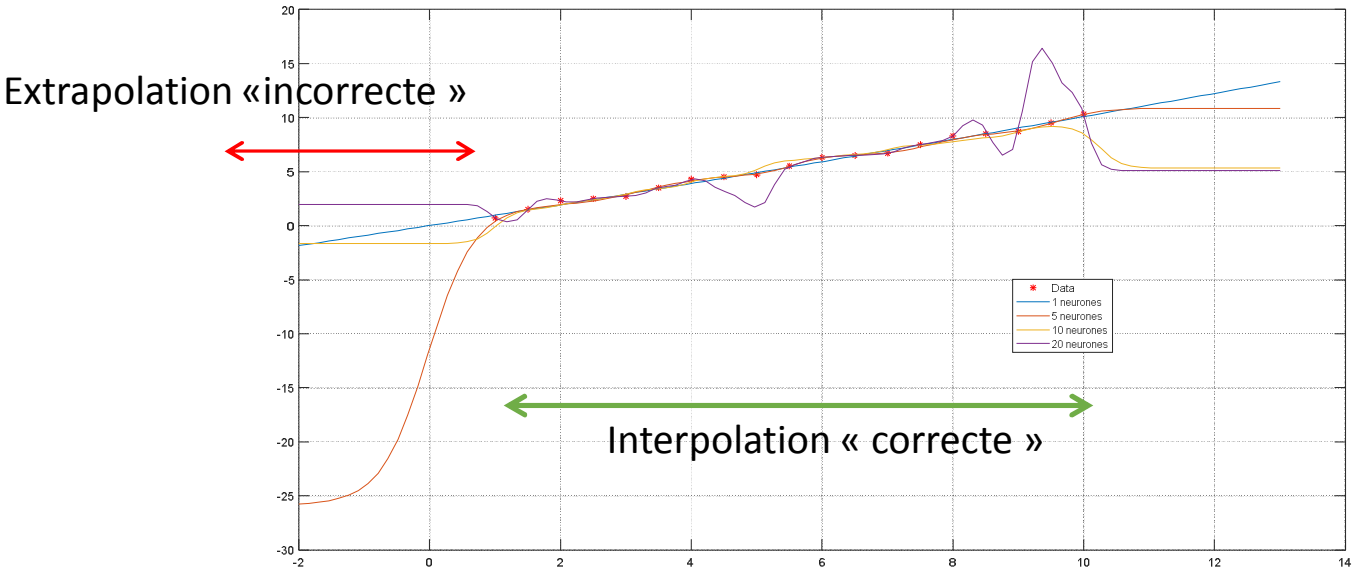
Réseau de neurones – *Apprentissage*

En général, on souhaite que le réseau donne de bonnes performances :

- Sur le jeu de donné utilisé pour l'apprentissage (évident)
- Sur des données inconnues: c'est la propriété de généralisation

La généralisation ne peut pas être parfaite: **Il n'y aucune raison pour que le réseau de neurones « crée » de l'information. Au mieux il peut interpoler correctement.**

Cependant, garantir une bonne interpolation, n'est pas un problème trivial, surtout pour des problèmes complexes.



Réseau de neurones – *Apprentissage*

Le problème théorique qui se cache derrière le phénomène d' 'overfitting' est que le problème ne contient pas assez d'information pour trouver la bonne valeur des poids

C'est un peu comme cherche à faire une régression polynomiale d'ordre 10 sur le jeu de données suivants:

X	3	4	5
y	2	8	3

=> le degrés du polynôme est trop important, il y a une infinité de solutions.
=> Beaucoup de ces solutions ne sont pas satisfaisantes

Le remède est simple: il faut donner à l'algorithme un moyen de choisir une « bonne » solution.
En général, les solutions avec les poids les plus petits sont souvent bonnes (en pratique, aucune garantie théorique)

$$C(W) = (1-\lambda)\sum_{i=1}^{n_x} \left\| \hat{y}(X_i, W) - y_i \right\| + \lambda \sum_{j=1}^{n_w} (w_j)^2$$

Si λ=0 alors il n'y a pas de régularisation
Si λ=1 alors on ne tient pas compte du critère, et le résultat est : $w_j = 0 \; _{j=1..n_W}$

Le problème est qu'en pratique, il n'y a pas de réelle méthode pour fixe la valeur de λ.
=> Méthode essais-erreur..

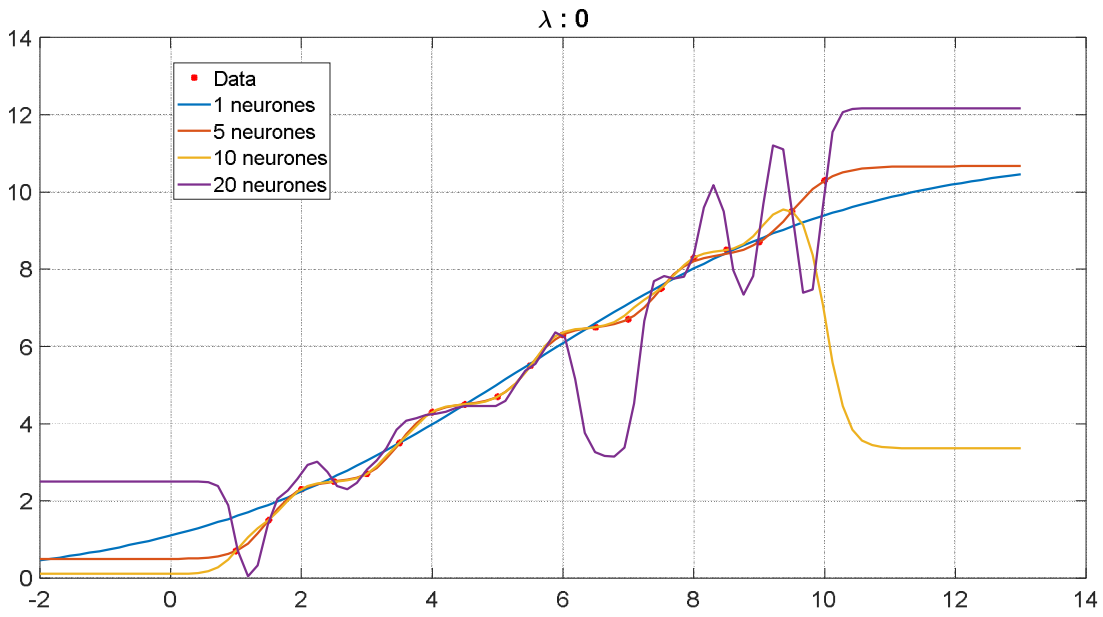
Réseau de neurones – *Apprentissage*

```
clear all;
close all;
clc;

% Démonstration du problème du réglage des hyper-paramètres
X=1:0.5:10;
f=@(x) x+.3*cos(3*pi*x);
Y=f(X);

figure;
plot(X,Y,'r*');
Lgd={'Data'};
hold on;
TailleHidden=[1 5 10 20];

for i=1:length(TailleHidden)
    clear net;
    net=feedforwardnet(TailleHidden(i));
    net.performParam.regularization=0.1;
    net.Layers{1}.transferFcn='tansig';
    net.Layers{1}.initFcn='initnw';
    net.divideParam.trainRatio = 1; % training set [%]
    [net] = train(net,X,Y);
    %view(net);

    % Visualisation de la capacité à interpoler & extrapoler
    dx=max(X)-min(X);
    X2=linspace(min(X)-dx/3,max(X)+dx/3,100);
    yest=net(X2);
    plot(X2,yest);
    Lgd{end+1}=sprintf('%i neurones',TailleHidden(i));
    legend(Lgd);
    title(sprintf('\\lambda : %.1g',net.performParam.regularization))
end
grid on;
```

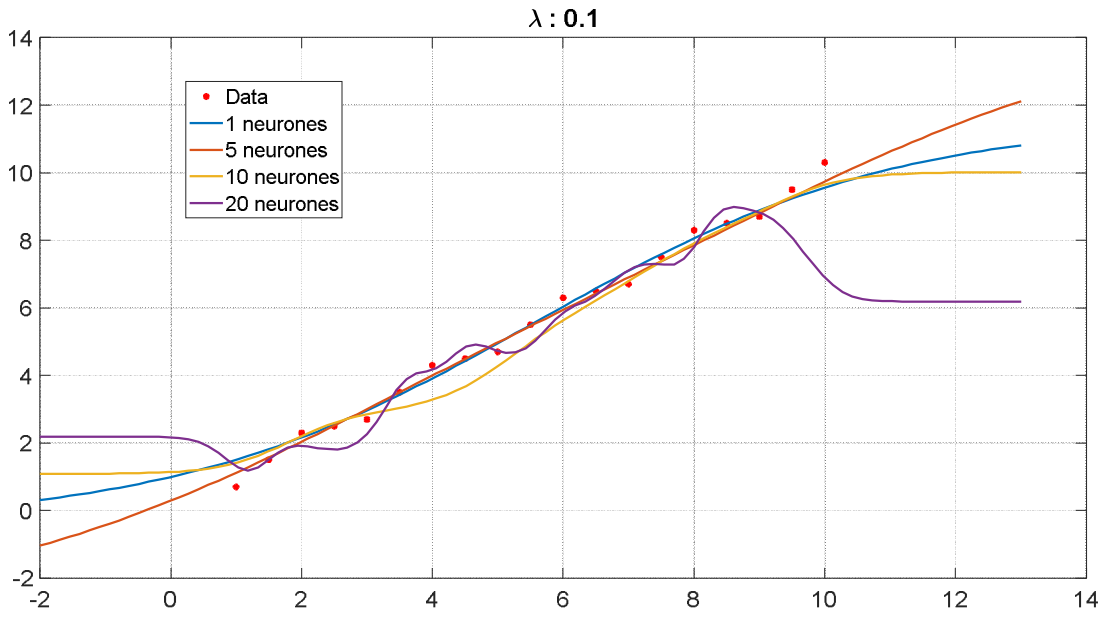
Taux de régularisation

Réseau de neurones – *Apprentissage*

Pas de régularisation : mauvais apprentissage des poids

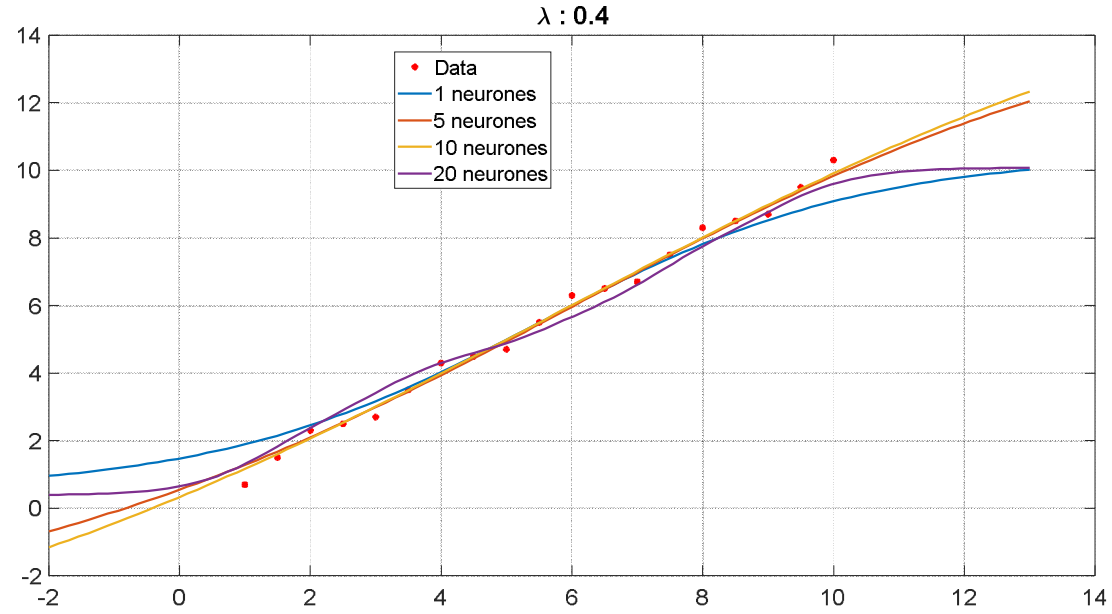


Avec régularisation: meilleur apprentissage des poids

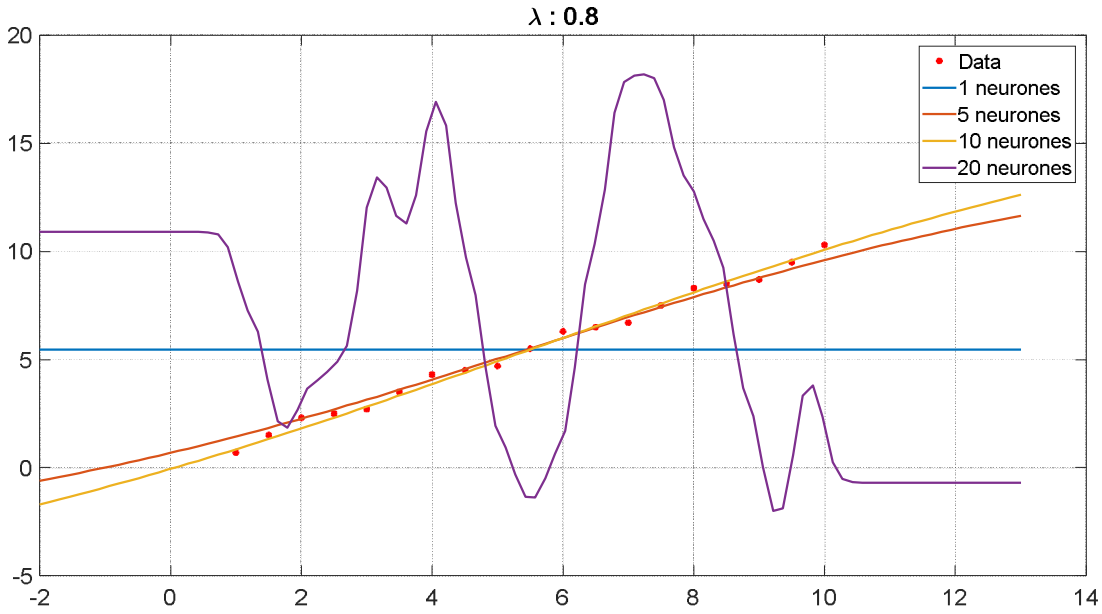


Réseau de neurones – *Apprentissage*

Réglage satisfaisant



Trop de régularisation: les données ne sont pas suffisamment prises en compte



Réseau de neurones – *Apprentissage*

Lorsqu'on dispose de beaucoup de données, on peut découper les données en plusieurs sous ensembles:

- Un jeu de données pour **l'apprentissage** des poids (valeur typique 70%). Normalement, l'erreur sur ce jeu de données doit descendre à chaque itération (principe d'un algorithme de descente)

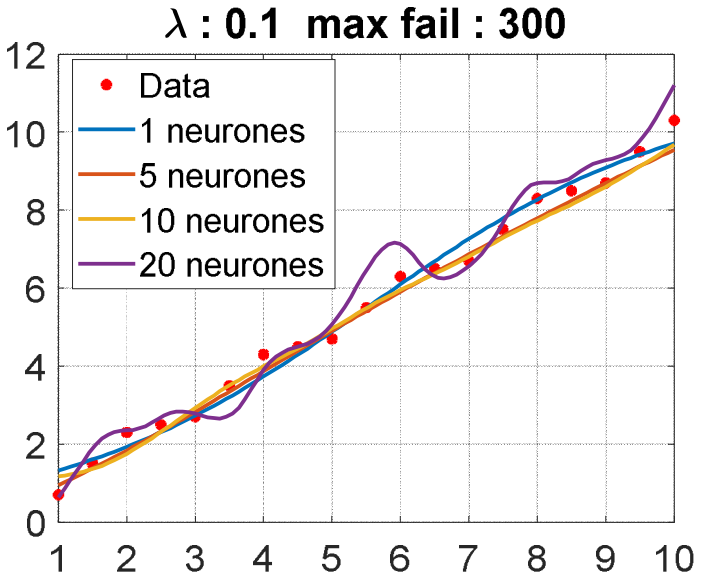
- Un jeu de données pour la **validation** (valeur typique 15%). Lorsque le réseau a tendance à « overfiter » l'erreur va être très faible sur les données d'apprentissage mais va augmenter sur les données de validation.
  => permet de mesurer la capacité du réseau à généraliser
  => on stoppe prématurément l'algorithme d'apprentissage si l'erreur sur le jeu de validation augmente alors que celle sur le jeu d'apprentissage continue à diminuer

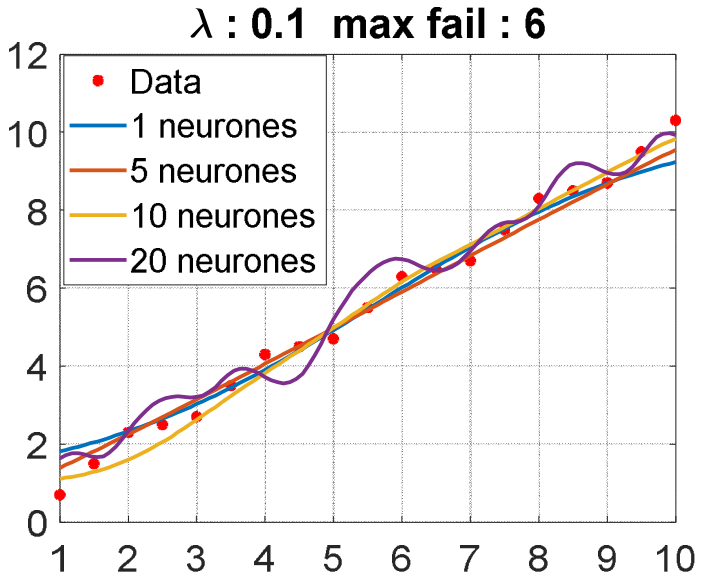
- Un jeu de données les **tests** (valeur typique 15%). Ces données permettent de comparer différents réseaux de neurones entre eux.

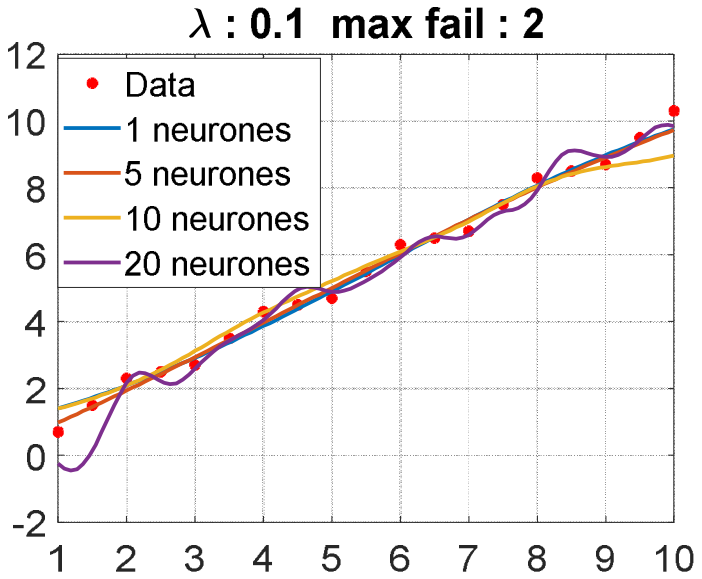
```
net.divideParam.trainRatio = 0.8; % training set [%]
net.divideParam.valRatio = 0.2; % validation set [%]
net.divideParam.testRatio = 0; % test set [%]
```

Réseau de neurones – *Apprentissage*

=> dans Matlab l'algorithme le paramètre « `net.trainParam.max_fail` » permet de spécifier le nombre maximum d'itérations avec augmentation du critère sur les données de validation.







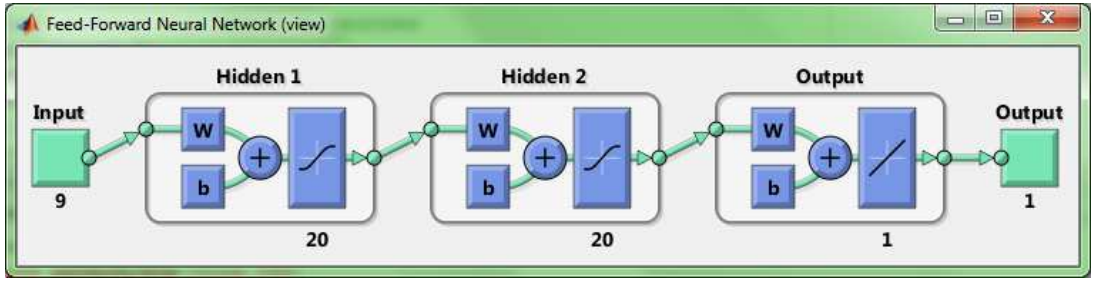
Réseau de neurones – *Différents types de réseaux*

Il existe différents types de structure réseaux:

- Feedforward

C'est l'architecture « classique »

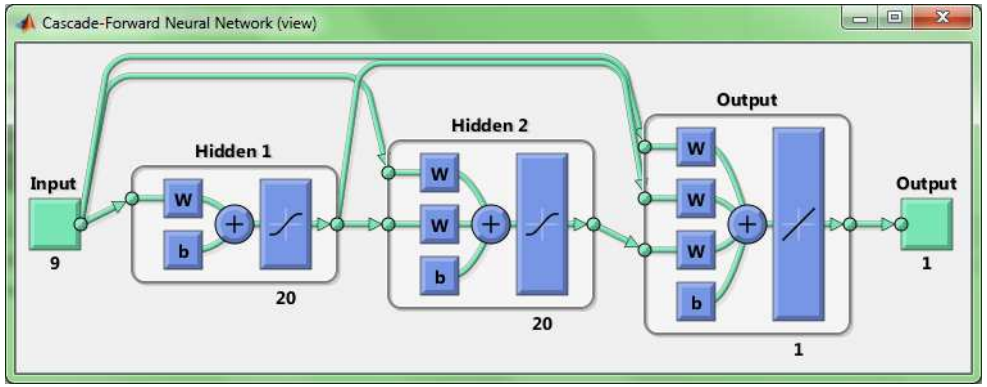
```
hiddenSizes=[20 20 ];
net = feedforwardnet(hiddenSizes);
```



- Cascade Feedforward

C'est l'architecture « classique » dans laquelle les entrées sont propagées à chaque couche cachées

```
hiddenSizes=[20 20 ];
net = cascadeforwardnet(hiddenSizes);
```



1) Réseau de neurones
2) Apprentissage des poids
3) Différents types de réseaux

Réseau de neurones – *Différents types de réseaux*

Lorsque l'on souhaite approcher le comportement d'un système dynamique, ou d'une série temporelle, il peut être utile de propager les valeurs précédentes de l'entrée et/ou de la sortie dans le réseau.

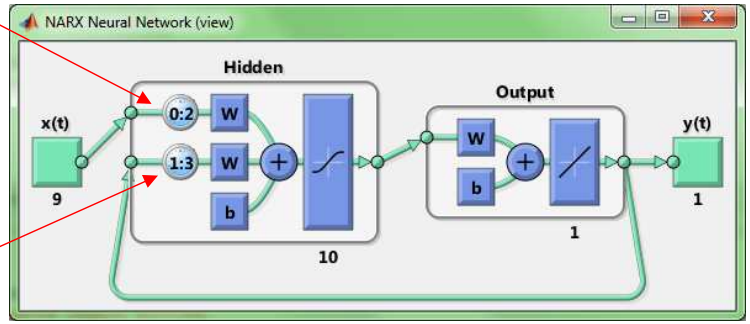
La simulation du réseau nécessite alors de fournir un état initial pour les valeurs retardées.

- NARX (Nonlinear autoregressive with external input)

```
inputDelays = 0:2;
feedbackDelays = 1:3;
hiddenLayerSize = 10;
net = narxnet(inputDelays,feedbackDelays,hiddenLayerSize,'open');
```

Retards sur l'entrée

Retards sur le bouclage de la sortie



La couche d'entrée est modifiée. Elle par une couche comprenant :

- Autant d'entrées retardées que de valeurs dans le vecteur `inputDelays`
- Autant de sorties retardées que de valeurs dans le vecteur `feedbackDelays`

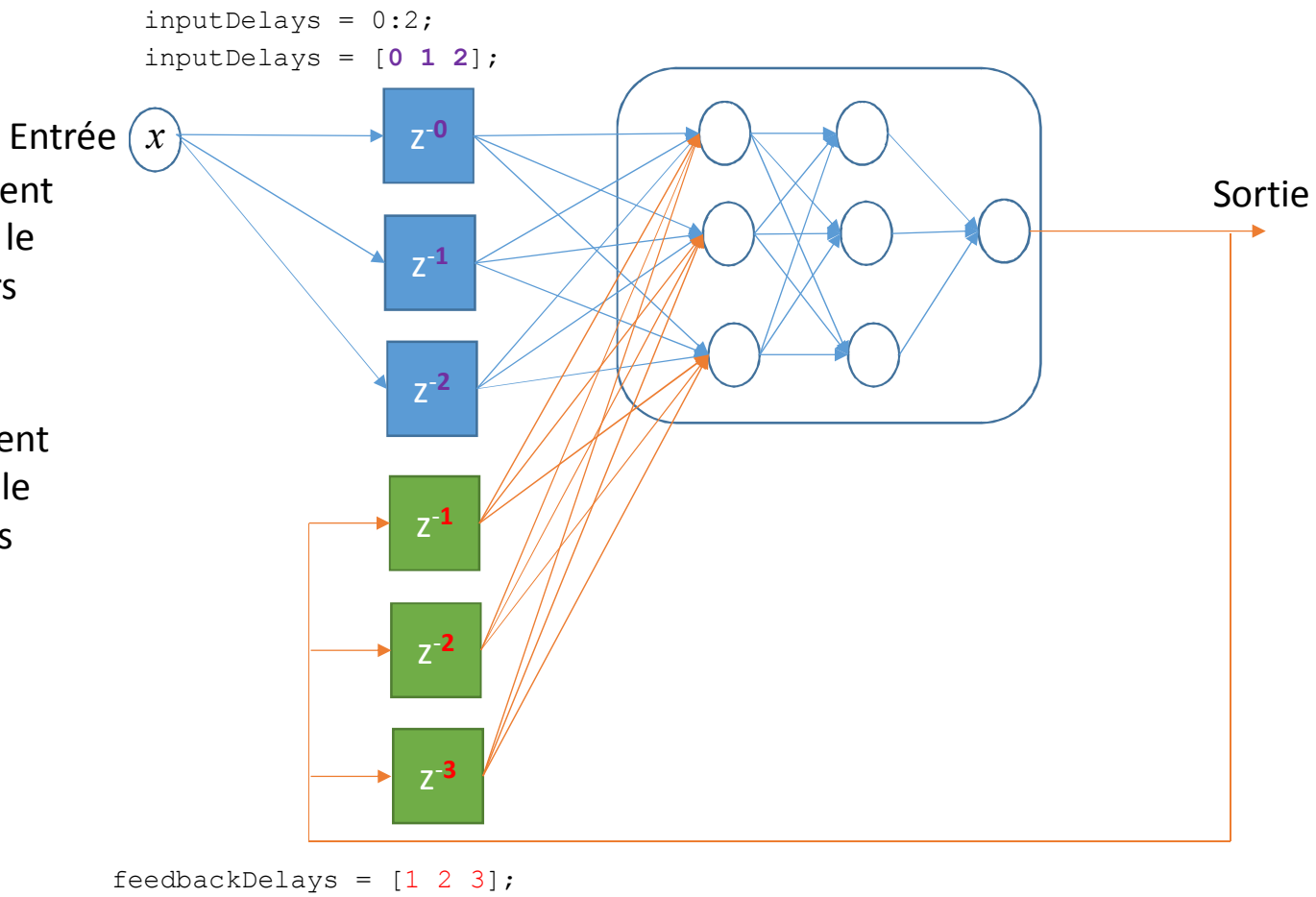
NB : evidemment, `feedbackDelays` ne peut pas contenir la valeur 0 (cela voudrait dire que la sortie qui n'est pas encore calculée alimente l'entrée)

Réseau de neurones – *Différents types de réseaux*

- NARX (Nonlinear autoregressive with external input)

NB : le réseau représenté ici ne contient qu'une entrée et qu'une sortie, mais le principe reste le même avec plusieurs entrées et/ou sorties

NB : le réseau représenté ici ne contient qu'une entrée et qu'une sortie, mais le principe reste le même avec plusieurs entrées et/ou sorties

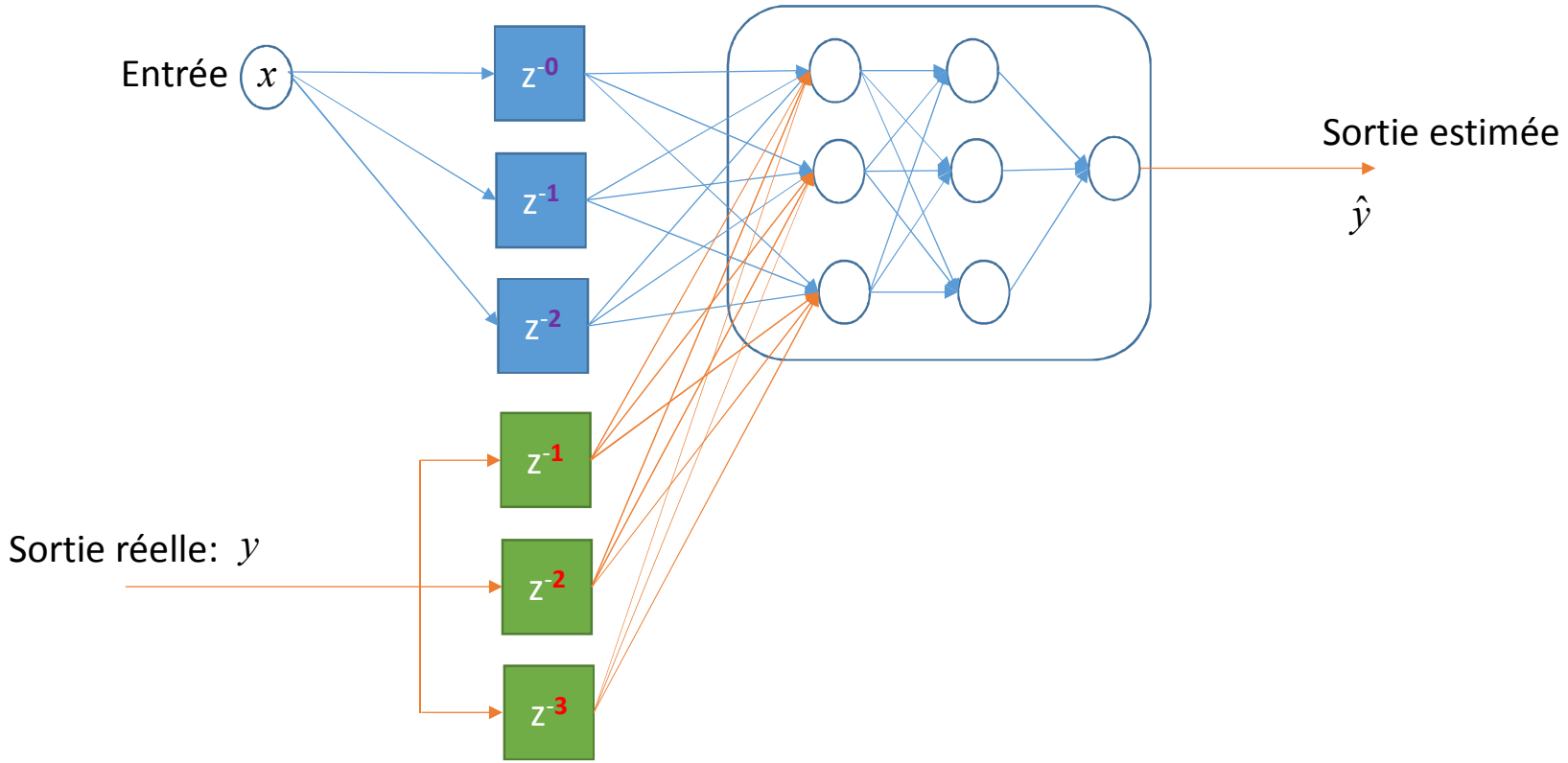


Réseau de neurones – *Différents types de réseaux*

- NARX (Nonlinear autoregressive with external input)

  La structure en boucle fermée de ce type de réseau peut rendre l'apprentissage très complexe.

  Cependant, il existe une solution relativement simple à mettre en œuvre: entrainer le réseau en utilisant comme sorties retardées non pas les sorties du réseau mais les valeurs cibles. Ainsi le réseau obtenu est un réseau en boucle ouverte.

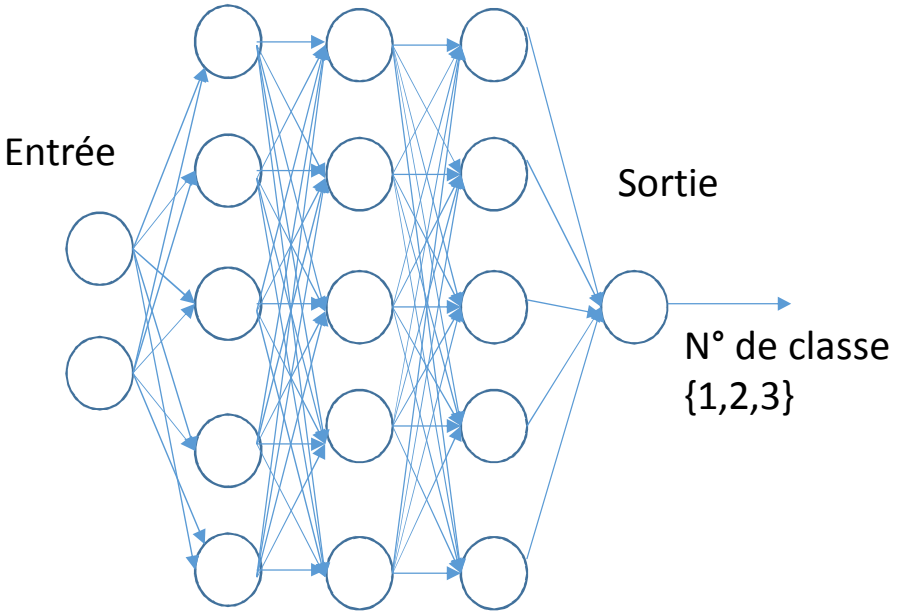


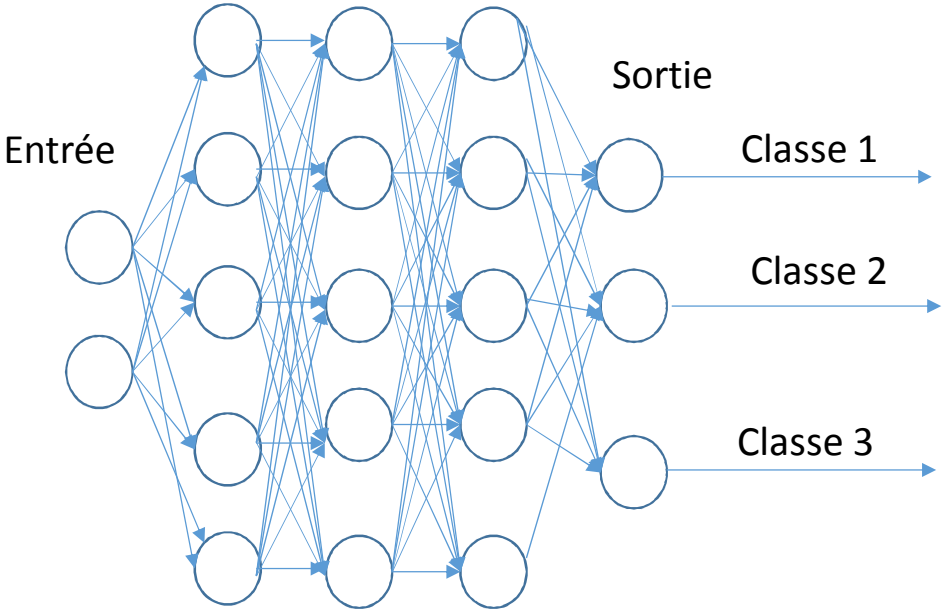
Réseau de neurones– *Classification avec une couche de sortie softmax*

Lorsqu'on cherche à faire une classification supervisée, le réseau doit apprendre à quelle classe appartient chaque image,

La sortie « utile » du réseau de neurones est le n° de la classe.

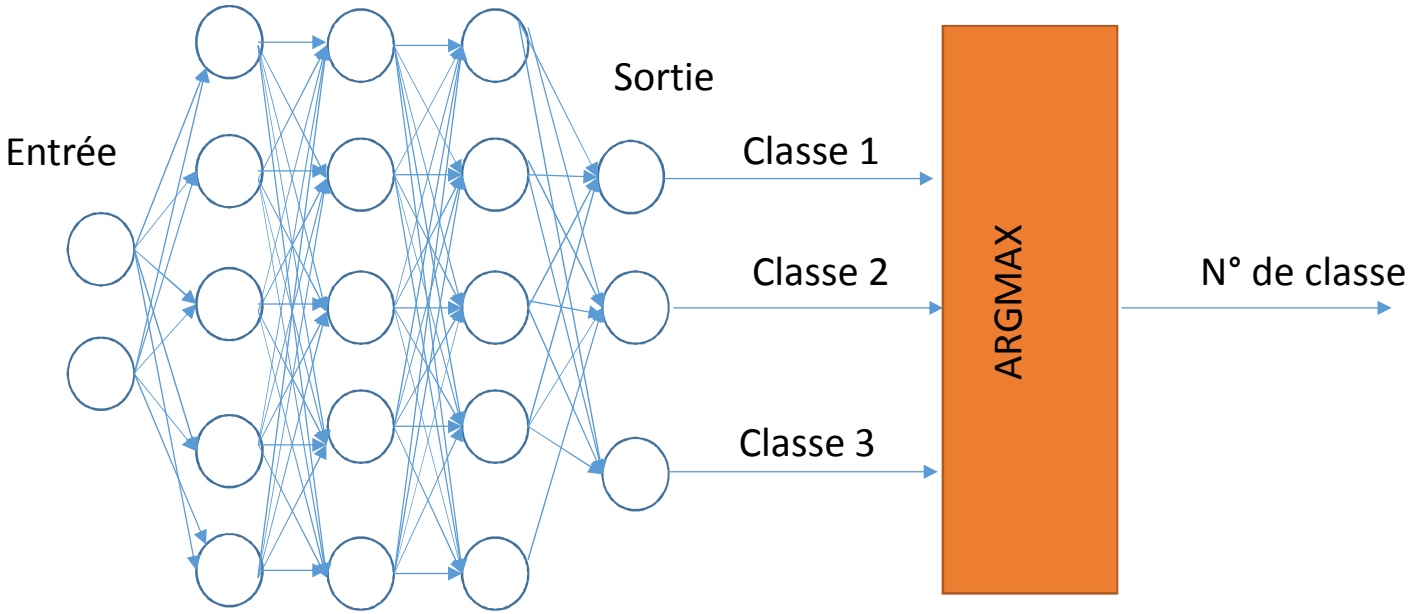
En pratique, on utilise plutôt un réseau de neurones avec autant de sortie que de classe





Réseau de neurones– *Classification avec une couche de sortie softmax*

Cependant, il faut prendre une décision sur quel est le n° de classe retenu car pour chaque entrée, le réseau génère 3 sorties.



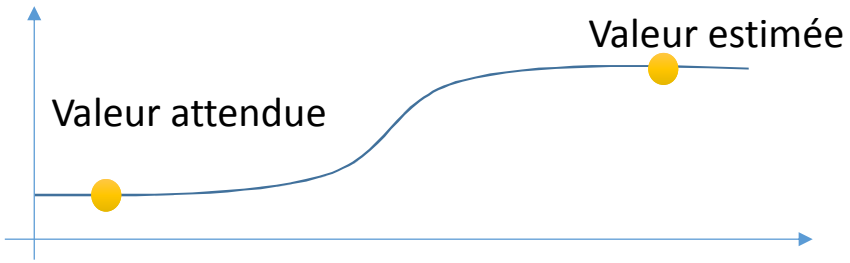
Problème : argmax n'est pas différentiable => impossible d'apprendre les poids avec une méthode à base de gradient

Réseau de neurones– *Classification avec une couche de sortie softmax*

Remède: Utiliser une sigmoïde car elle est bornée entre 0 et 1

$$E = \sum_{i=0}^{Nimages} \sum_{j=1}^{Nclasse} \left(y_i^j - \hat{y}_i^j \right)^2 = \sum_{i=0}^{Nimages} \sum_{j=1}^{Nclasse} \left(y_i^j - \hat{y}_i^j \right)^2$$

Re-problème : Lorsque la classification est incorrecte, le point de fonctionnement se trouve sur la partie « plate » de la sigmoïde:



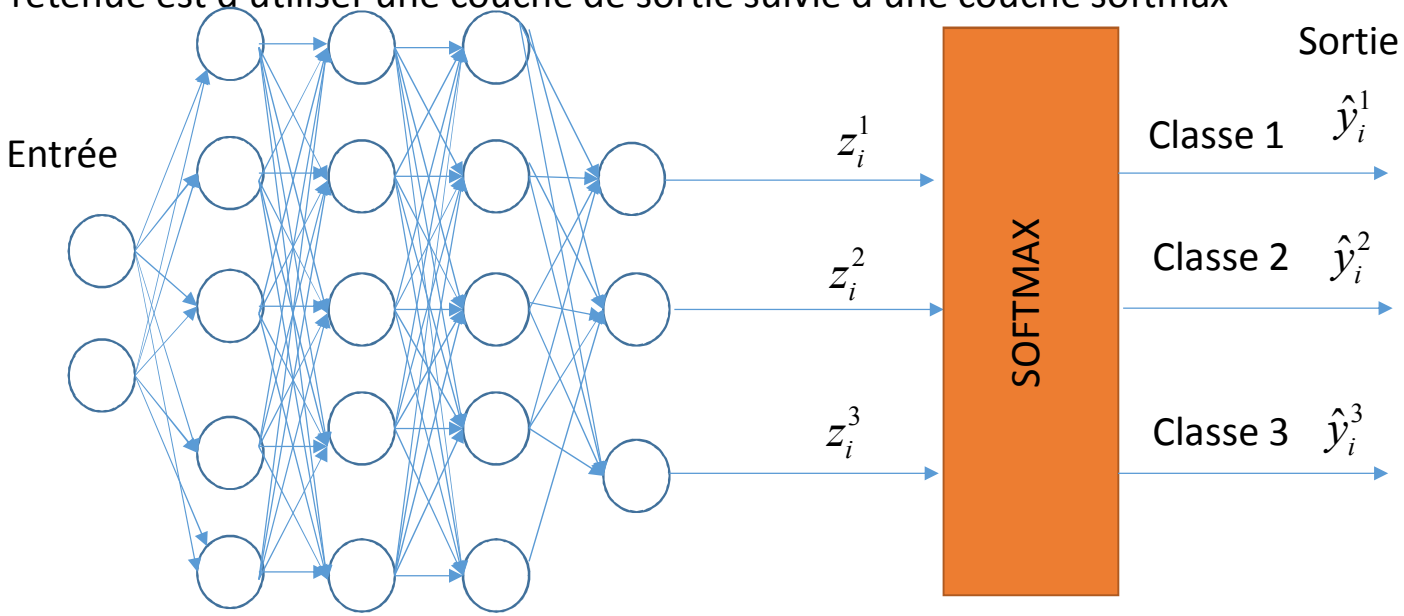
Dans les zones recherchées, la sigmoïde est plate
⇒ Le gradient de l'erreur est quasi nul
⇒ Rétropropagation peu efficace

En outre, on sait qu'une seule classe est activée à la fois: $\sum_{j=1}^{Nclasse} y_i^j = 1$

Le réseau de neurones n'intègre pas cette information.

Réseau de neurones– *Classification avec une couche de sortie softmax*

La solution retenue est d'utiliser une couche de sortie suivie d'une couche softmax



$$\hat{y}_i^j = \frac{e^{z_i^j}}{\sum_{k=1}^{Nclasse} e^{z_i^k}}$$

L'intérêt est qu'avec la normalisation, l'information est bien stockée dans le classement des valeurs u_i^j qui est important au lieu de leur valeurs individuelles.

L'opérateur « softmax » est différentiable => possibilité d'utiliser les méthodes de rétropropagation

Les sorties peuvent être interprétées comme des probabilité que l'entrée appartienne aux différentes classes.

Les sorties du réseau vérifient: $0 < \hat{y}_i^j < 1$ $\sum_{j=1}^{Nclasse} \hat{y}_i^j = 1$

Réseau de neurones – *Classification avec une couche de sortie softmax*

Le critère utilisé n'est pas l'erreur quadratique mais l'entropie croisée:

$$E = -\sum_{i=1}^{n} \sum_{j=1}^{Nclasses} y_i^j \ln\left(\hat{y}_i^j\right)$$

L'erreur est toujours positive et le critère est dérivable.

L'estimée doit vérifier la propriété de somme convexe: $\sum_{j=1}^{Nclasse} \hat{y}_i^j = 1$

$$E = -\sum_{i=1}^{n} \sum_{j=1}^{Nclasses} y_i^j \ln\left(\frac{e^{u_i^j}}{\sum_{k=1}^{Nclasse} e^{u_i^k}}\right) = -\sum_{i=1}^{n} \sum_{j=1}^{Nclasses} \left[y_i^j \left(\ln e^{u_i^j} - \ln \sum_{k=1}^{Nclasse} e^{u_i^k} \right) \right]$$

$$E = -\sum_{i=1}^{n} \sum_{j=1}^{Nclasses} \left[y_i^j \left(u_i^j - \ln \sum_{k=1}^{Nclasse} e^{u_i^k} \right) \right]$$

Réseau de neurones– *Classification avec une couche de sortie softmat*

Le critère utilisé n'est pas l'erreur quadratique mais l'entropie croisée: $E = -\sum_{i=1}^{n} \sum_{j=1}^{Nclasses} y_i^j \ln\left(\hat{y}_i^j\right)$

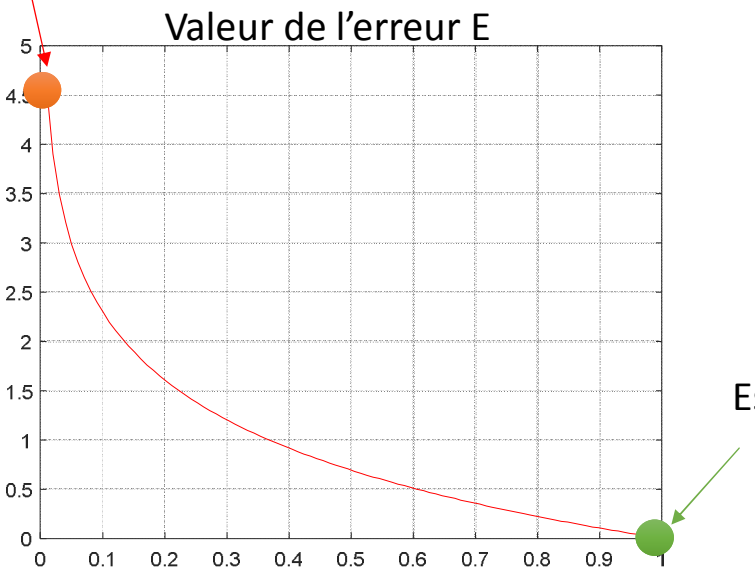
$\Rightarrow$ Le gradient de l'erreur est très important lorsque l'estimation est fausse

$\Rightarrow$ Apprentissage facilité

Exemple : 2 classes

$y_i^1 = 1, y_i^2 = 1$

Estimation incorrecte



Estimation correcte

$\hat{y}_i^1 = 0$

$\hat{y}_i^2 = 1$

$\hat{y}_i^1 = 1$

$\hat{y}_i^0 = 1$